April 2013

# Educational RTOS Development Board

Nicholas Anthony DeMarinis
*Worcester Polytechnic Institute*


# Educational RTOS Development Board

*A Major Qualifying Project Report submitted to the Faculty of*
*WORCESTER POLYTECHNIC INSTITUTE*
*In partial fulfillment of the requirements for the*
*Degree of Bachelor of Science*

Submitted By
Nicholas DeMarinis
ndemarinis@wpi.edu

Advised by
Professor Hugh Lauer
lauer@wpi.edu

Professor Fred Looft
fjlooft@wpi.edu

**Submitted 25 April 2013**

# Abstract

The objective of this project was to facilitate student learning of embedded systems design. At WPI, students in ECE3849 must combine hardware and software concepts to develop real-time embedded systems in labs, a process which often frustrates students. This project identified ways to engage students in embedded systems design by 1) identifying ECE3849's educational objectives 2) designing a versatile peripheral board to support new labs, 3) synthesizing student feedback on their frustrations and 4) developing targeted documentation for students to help alleviate their frustrations in labs. My development board, documentation, and critical analysis of student feedback provide recommendations for instructors to help future offerings of ECE3849 challenge students to design embedded systems.

# Table of Contents

# List of Figures

# List of Tables

# 1. Introduction

Embedded systems are computers inside other products, or computer systems dedicated to performing a set of tightly coupled functions to accomplish tasks. Embedded systems encompass a very wide range of applications, including devices we use every day as well as sophisticated, application-specific systems. Figure 1 shows some example embedded systems: a simple device like a digital alarm clock embeds a small microprocessor (often called a microcontroller) to perform very specific tasks like accurately recording the time and controlling a display. Many embedded systems perform a variety of functions, such as graphing calculators that can perform an astounding array of mathematical and scientific functions. Similarly, a modern automobile incorporates over 50 microcontrollers to perform tasks like managing engine emissions, controlling the radio, and operating anti-lock brakes (Movtalli, 2010). Embedded systems like these are integral to so many applications: it is estimated that embedded microprocessors account for 98% of the integrated circuits produced each year (Ebert & Jones, 2009, p. 42). Thus, embedded systems represent an important segment of computing devices in our modern world.


**Figure 1: Examples of everyday embedded systems**

## Embedded Systems and Software Engineering

Embedded systems integrate with so many applications by controlling a device's specific hardware with software, bridging the gap between hardware and software design. This allows system designers to integrate the computational power and flexibility of software with a product's hardware to perform a task. In addition, many embedded systems have strict real-time performance requirements or must perform large numbers of complex tasks. To handle these requirements, some embedded systems utilize real-time operating systems (RTOSes), which provide software abstractions for running many tasks like a general purpose operating system, but

with a specific focus on meeting real-time deadlines (Wind River, 2011). In this way, modern embedded systems inherently include many of the same software engineering challenges of software systems. Thus, embedded systems design requires the integration of both hardware and software engineering to perform tasks.

## Teaching Embedded Systems

When studying embedded systems, students are challenged to integrate their knowledge of software development and hardware design to create a working system. The Electrical Computer Engineering (ECE) department teaches embedded systems through two courses: ECE 2049 and ECE 3849. ECE 2049 introduces students to basic embedded and system-level programming concepts. ECE 3849 challenges students to design more advanced embedded systems with real-time programming constraints. Most notably, ECE 3849 introduces multitasking concepts like scheduling, synchronization primitives, and real-time operating systems, all of which are high-level computer science topics. This is a significant step to ensuring that students studying computer engineering, computer science, and robotics are introduced to the software challenges of modern embedded systems.

The current offerings of ECE 3849 use an off-the-shelf development board to support the course's laboratory assignments. While this provides a formidable solution to demonstrating the software concepts taught in the course, the ECE department wants to investigate how a custom hardware platform could improve student learning in the course. In addition, the ECE department needs to find a way to teach this very diverse range of concepts without overwhelming students. In both ECE 2049 and ECE 3849, instructors often cite that students encounter difficulty when faced with the unique challenges of embedded programming, which are very different from their introductory programming courses. Students often struggle with identifying how they can integrate the hardware concepts they learn in lecture into their software designs. Since software concepts are so integral to ECE 3849 in order to meet the demands of modern embedded systems, it is critical that students learn these techniques as part of their embedded systems education. In doing so, embedded systems labs should excite students about the challenges of embedded programming, rather than frustrate them, and ensure that they develop a strong background for applying these concepts to their other courses and in a professional environment.

## Project Statement

The purpose of this project was to design and implement a comprehensive development platform to facilitate student learning of embedded programming techniques. This involved not only designing hardware for ECE labs, but also identifying how ECE instructors can integrate students' hardware *and* software backgrounds into ECE 3849. Therefore, my project was focused on designing hardware and making the software resources for the course accessible and usable by students. In order to accomplish this goal, I 1) identified requirements for facilitating student learning in ECE 3849, 2) designed and built a development board based on these requirements,

3) demonstrated how the development board can be used in labs and provide materials for instructors to use it in future laboratory assignments in the course, and 4) provided recommendations for instructors to guide students as they understand the hardware and software concepts they learn in labs. To do this, I developed a hardware development board targeted to ECE instructors and students requirements for deployment in ECE 3849's labs and developed an extensible software library to demonstrate its hardware. As a tutor for ECE 3849, I analyzed students' frustrations with the course material and created a set of targeted documentation—a "cookbook" of step-by-step instructions, and a more accessible software environment to help alleviate these specific challenges. My development board and documentation provide a set of recommendations and materials that instructors can use to help evolve future offerings of embedded systems courses at WPI.

## Report Summary

Chapter 2 of this report provides a background on embedded systems and the embedded systems curriculum at WPI, including an overview of the hardware and software concepts necessary for these courses and how students apply them in ECE 2049 and ECE 3849. Chapter 3 describes my design process for creating the hardware development board. Chapter 4 describes the development board's key requirements, as well as its system architecture, implementation details, and hardware testing results. Chapters 5 and 6 describe my process for identifying students' frustrations as a tutor for the course and outline the key frustrations I observed and how they led to the development of a "cookbook" of documentation. Finally, chapter 7 concludes with a set of recommendations for instructors and future goals to help facilitate student learning of embedded systems.

# 2. Background

This section outlines the specific challenges of students learning embedded systems at WPI and reviews current methods for alleviating these challenges in embedded systems curricula. As described in the previous section, designing a development system for ECE 3849 is a difficult problem due to the diversity of course topics inherent to embedded systems. In order to study this problem more fully, I identified the following research questions:

- What are the educational objectives for effective teaching and learning of embedded systems?
- What are the pedagogical concerns associated with teaching embedded systems courses?
- How does WPI's embedded systems curriculum address these concerns?

This section outlines the challenges inherent to embedded systems education and maps these concerns to the existing curriculum at WPI. From there, I could identify any gaps in the existing curriculum that my design could help alleviate on a hardware or software level. This helped me determine how my design could best help students develop comprehensive knowledge of embedded systems in their courses.

## 2.1. What are embedded systems?

Modern embedded systems integrate complex hardware and software in order to accomplish specific tasks. Therefore, embedded systems courses need to incorporate both hardware and software concerns into their curricula in order to ensure their students meet the evolving demands of these systems. A simple example for describing the integration required in an embedded system is a digital alarm clock, like the one shown in Figure 1. Such a system might contain the following devices:

- A display to show the time to the user
- A speaker or buzzer that produces the alarm
- Buttons for the user to set the time and alarm

This system requires a fairly straightforward hardware design, as it includes very few components. The software implementation for such a device is also very straightforward: its few functions could be incorporated into a single control loop or state machine. In fact, a simple design like this is often a starting point for many introductory embedded systems (Davies, 2008; Kamal, 2006).

However, many embedded systems have more demanding requirements: many devices are often required to perform a large number of complex tasks, or complete certain tasks within a certain time period in order to maintain reliability or performance requirements. NASA's latest rover *Curiosity*, shown in Figure 2, is an example of such a system, as it has a very diverse set of

objectives: it must navigate on another planet using sensor data, operate a number of sophisticated hardware instruments, and communicate with Earth to send useful data and receive instructions (JPL, 2013). To carry out these objectives, *Curiosity*'s main computer uses only 200 MHz processor and 256 MB of RAM, yet it must perform a staggering number of tasks in order to maintain its systems in a harsh environment, conduct scientific experiments, and relay information back to Earth.

Designing an advanced embedded system like *Curiosity* requires a complex integration of hardware and software to accomplish its objectives. According to the rover's developers, *Curiosity*'s software comprises over 2.5 million lines of C code that runs in over 130 active threads and was developed on a team of over 40 developers and testers (Havelund, Groce, Smith, & Barringer, 2009). In this way, modern embedded system design includes the same software engineering challenges as other large software-based projects. This underscores that embedded system design is a significant field of computing.

**Figure 2: NASA's Curiosity Rover**

In addition, embedded systems design entails a number of domain-specific design challenges, including the complex integration of hardware and software requirements to accomplish tasks. *Curiosity* must process data from its instruments and sensors in "real-time" in order to navigate and manage its systems effectively. Real-time operation is a fundamental requirement for many systems that operate autonomously, including robots like *Curiosity*, guidance systems, or video

and audio processing applications. This introduces a set of software design requirements not found in most computer science fields.

To accomplish these real-time objectives and support such a multithreaded system, *Curiosity* uses a real-time operating system (or RTOS) to manage execution of its tasks. An RTOS provides the software abstractions, memory management, and synchronization primitives necessary for a multitasking environment—similar to a general purpose operating system—but with a focus on meeting real-time performance constraints. *Curiosity* uses the VxWorks RTOS, which includes a monolithic kernel, networking stack, and a host of software libraries to facilitate development of complex embedded systems (Miller, 2012; Wind River, 2011).

In this way, the development of embedded systems is far more complex than a simple alarm clock—modern embedded systems design includes significant components of both digital system design and software development in professional environments. Embedded systems development represents an important intersection point between computer engineering and computer science that uniquely challenges students to apply their skills in both fields. This requires the integration of software and hardware engineering curricula to effectively teach students how to develop embedded systems as part of their education and in a professional environment.

## 2.2. How does embedded systems education fit into the Digital ECE curriculum at WPI?

The Digital ECE curriculum currently encompasses five courses, which allow students to explore the concepts, design, and applications of both digital logic and embedded systems at the sophomore, junior, and senior levels. To receive an ECE major, all ECE students are required to take at least two of these courses to understand the basics of digital systems, how they are utilized in countless applications in today's world, and how to design and interface with them in a professional environment. The current course chart is shown in Figure 3.

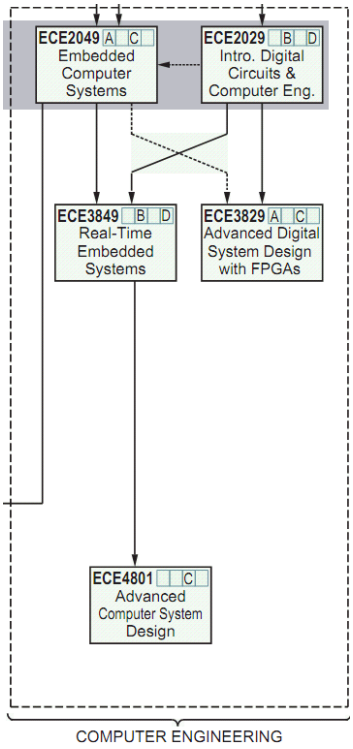



**Figure 3: Computer Engineering Course Chart as of A-Term 2012 ("Undergraduate Courses," 2012)**

Two introductory courses are offered at the sophomore level, ECE 2029 and ECE 2049, to teach students the fundamentals of digital logic and embedded systems, respectively. Students who wish to persue these fields can take the two junior-level courses, ECE 3829 and ECE 3849, which teach students to design digital systems based on requirements in their laboratory experiments while exploring the underlying architecture of logic and embedded systems, respectively. Finally, ECE 4801 provides students an opportunity to study the architecture of digital systems in much more detail, providing a basis for the graduate courses in computer architecture. Thus, the entire progression of digital ECE courses aims to provide a comprehensive understanding of computer system design, from application to system architecture. For reference, the full course descriptions are provided in Appendix A.

In the current curriculum, both ECE 2049 and ECE 3849 specifically teach students the fundamental concepts and design of embedded systems. ECE 2049 defines embedded systems and explains how they are so prevalent in our world, while explaining fundamental concepts like timers, interrupts, and analog-to-digital conversion and demonstrating them in lab by using simple peripheral devices. This provides a high-level overview of embedded systems development and introduces students to resource-limited programming, which is critical for more complex systems. Since embedded systems are used in so many applications, ECE 2049 also caters to a very wide audience: it is taken by ECE majors interested in digital systems, majors who are persuing other concentrations and need the requirement, as well as many Robotics Engineering (RBE) majors. Regardless of their field, the course provides students the

opportunity to program embedded systems on a high level and learn to utilize the fundamental tools a microcontroller provides.

ECE 3849 takes this a step further by exploring the architecture and design of advanced embedded systems, also adding the real-time element as a basis for discussion of more advanced course topics. ECE 3849's course description explains that its main focus is to "solve real-world problems that require an embedded system to meet strict real-time constraints with limited resources." Based on this, students use the concepts introduced in class to develop a system in lab for specific design goals. The description goes on to note that the course continually introduces I/O standards and peripheral devices to help meet these goals; these include the study of memory types and interfaces, serial and parallel peripheral buses, DMA, and many others—the full list of concepts is described in Appendix A. ECE3849 also explores an equal number of software design concepts like multitasking, real-time scheduling, and eventually real-time operating systems ("Undergraduate Courses," 2012). This not only requires some familiarity with the hardware architecture of the embedded system that students use in lab, but it also entails that students strictly design their software implementations to meet these requirement, meaning that ECE 3849 requires that students integrate hardware and software engineering concepts to design embedded systems in lab.

Since hardware and software development are fundamental components of both ECE 2049 and ECE 3849, it is recommended that students taking these embedded systems course have a background in digital systems design and Computer Science (CS) topics before starting the embedded systems curriculum. The "recommended background" courses for ECE 2049 and ECE 3849 comprises three courses, two of which are CS courses, as shown in Figure 4. For reference, the full course descriptions for the listed CS courses as of this year are included in Appendix B.

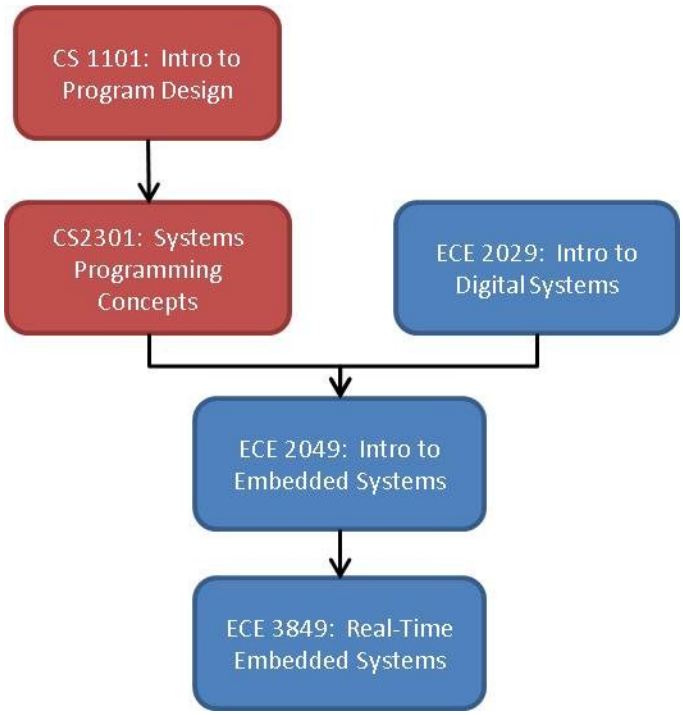



**Figure 4: Recommended Background for ECE 2049 and ECE 3849**

ECE 2029 covers the fundamental digital systems concepts necessary for understanding the hardware components of ECE 2049 and ECE 3849, including digital representations of data, peripheral devices, and registers. Students begin learning programming techniques in CS 1101, which introduce students to program design techniques and data structures. Students expand on these concepts in CS 2301 or CS 2303 to learn systems programming techniques, which includes learning how to develop and debug programs in C. Students in CS 2301 learn the basics of C programming, including pointers and memory management, data structures, the use of the debugger, and the execution stack ("Undergraduate Courses", 2012b). These programming and debugging concepts are critical to ECE 2049 and ECE 3849, as they provide the basis for students to implement their laboratory assignments. The set of concepts provided by these background courses and those required by the ECE 2049 and ECE 3849 is shown in Figure 5.

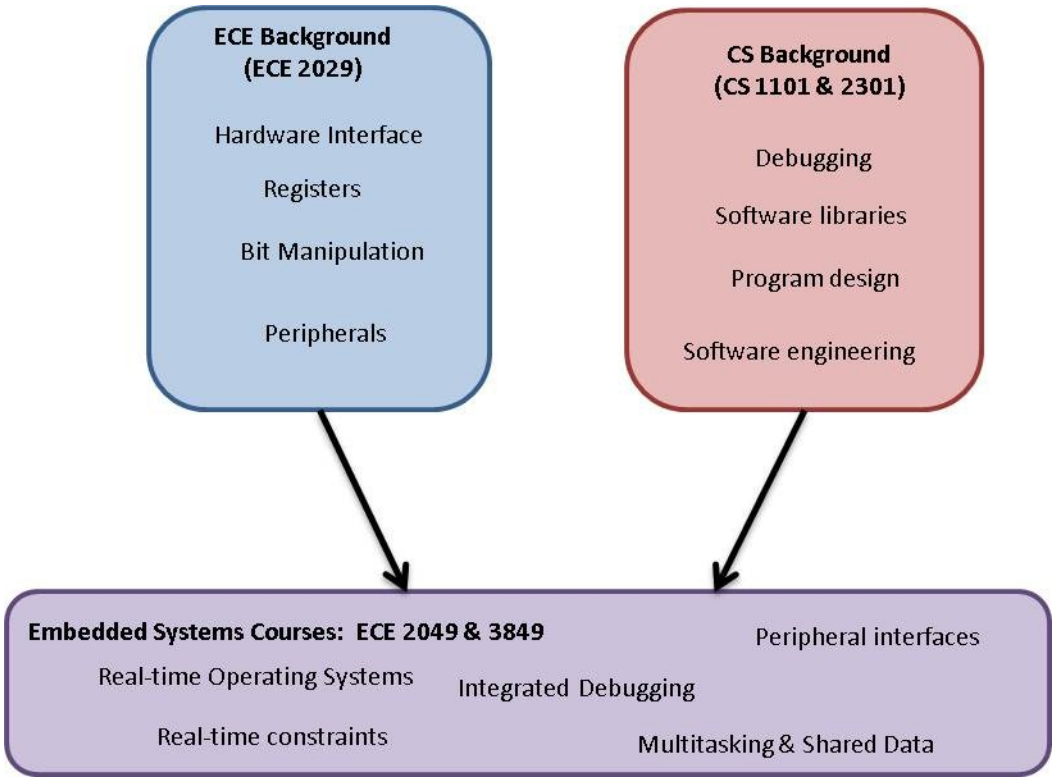



**Figure 5: Embedded Systems Course Concept Map**

However, embedded systems courses like ECE 2049 and ECE 3849 require a much more comprehensive understanding of the basics of program design and debugging to accomplish their course objectives. As discussed in section 2.1, embedded systems design entails many advanced software engineering challenges for its developers including large codebases, software libraries, and testing of hardware and software. When combined with the domain-specific challenges of embedded systems design such as real-time constraints, integrated debugging, and real-time operating systems, these courses are inordinately more complex than students' introductory CS and ECE courses.

In response to this, students in ECE 3849 often express frustration with their insufficient backgrounds in software and hardware design when implementing their labs. Most notably, their frustrations are centered on using the software development tools in the course, including the integrated development environment (IDE) that students use to write their code, the debugger, and their confusion with the libraries of existing code that they use in their labs. While these concepts are briefly taught in students' introductory CS courses, they do not cover these concepts to the extent necessary for use in a complex embedded systems course. Since students are ill-prepared for the software challenges, embedded systems courses need to teach these concepts as part of their course curriculum to ensure that students have sufficient background in the software development concepts to develop their labs. Thus, embedded systems courses like ECE 3849 require that students integrate their knowledge of hardware and software concepts in order to apply it to embedded systems design.

## 2.3. How do ECE students use embedded systems in their labs?

While students learn the basics of the concepts for both ECE2049 and ECE3849 in lecture, much of each course's time is devoted to students' ability to understand and utilize these concepts in lab. Typically, students are given a lab assignment with a set of design goals for a particular design or functional goal (ex. Use the temperature sensor to do X) with one or two weeks to implement the design, demonstrate it to the course staff, and write a comprehensive report to justify their implementation. Each course allocates three hours per week for students to work on their assignments in lab while receiving instruction from the course staff; however, these assignments require that students spend much more time outside of class—at least 10-12 hours per week—to implement their assignment. In this way, a large portion of a student's time in an embedded systems course is devoted to labs, allowing students a lot of time to demonstrate and practice the concepts they learn in lecture.

In labs, students work with real embedded systems to develop their lab assignments, which give students the same "hands-on" experience they might find in a professional embedded development environment. Typically, students work in teams of two at one of 25 lab stations in AK113, which is an academic lab specifically designated for digital and embedded systems courses. Each lab station is provided an embedded system for students to program and test their implementations throughout the course; in addition, each station includes fundamental tools for debugging and prototyping with external devices and peripherals: a logic analyzer, power supply, and digital multimeter. A typical lab station is shown in Figure 6 below.

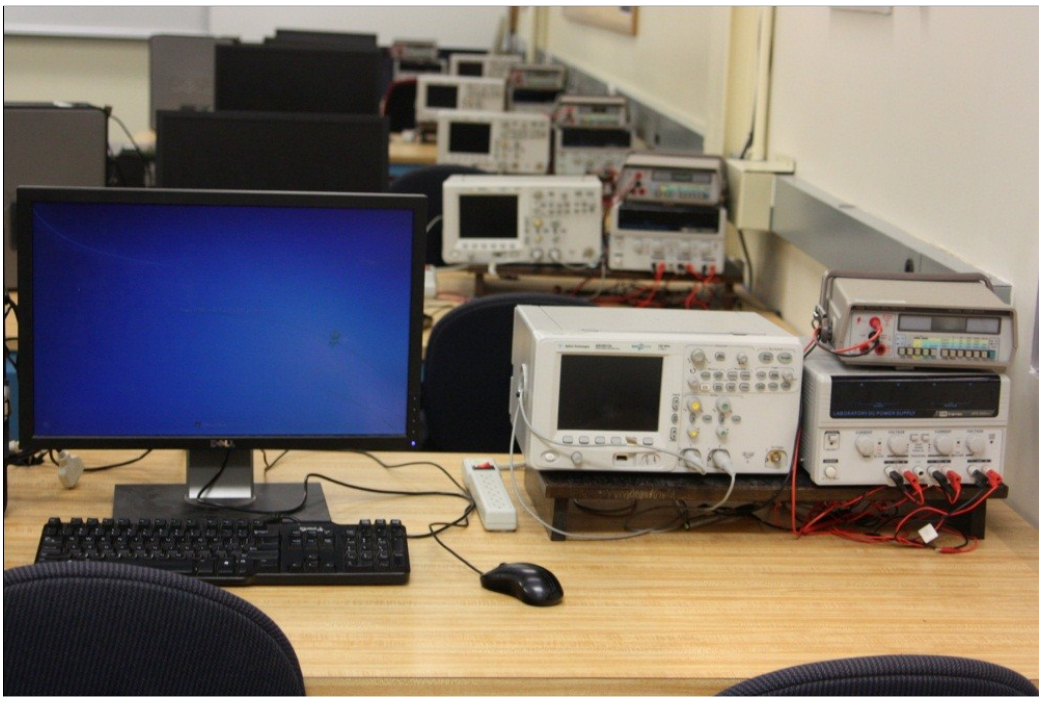


**Figure 6: Student lab station in AK 113**

Students' embedded lab experience begins in ECE2049. In this course, students learn basic embedded development concepts like timers and interrupts while interfacing with peripherals made available on a development platform. The platform consisted of two components: an Olimex educational development board containing a TI MSP430 microcontroller, an LCD and buttons as well as an expansion board designed by a WPI student specifically for the course—the entire platform is shown in Figure 7 ("MSP430 Expansion Board").

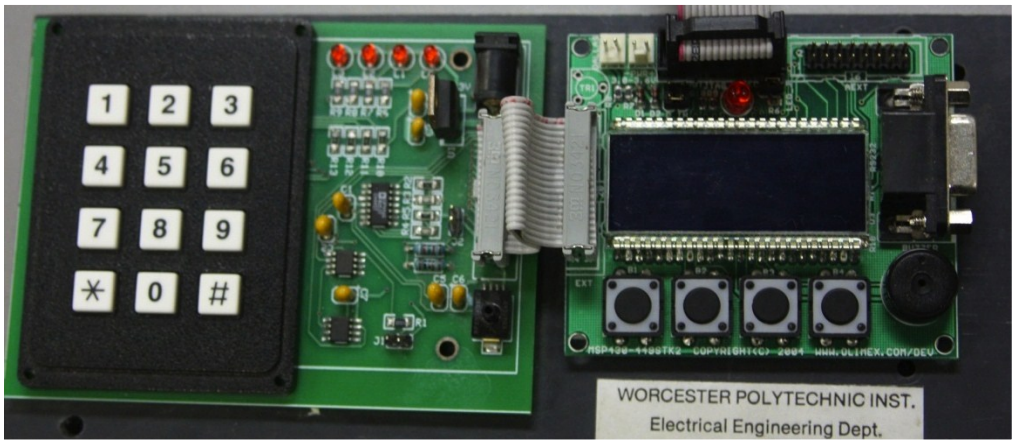



**Figure 7: MSP430-based Development Platform for ECE2049**

The development platform for ECE2049 contains the following peripherals for student labs:

- **User Input/Output**: 7 Character LCD, Keypad, 4 buttons, 4 LEDs, Buzzer
- **Sensors**: Temperature, Pressure
- **Signals**: Analog inputs, Digital-to-Analog converter (DAC)

Using these peripheral devices, students learn fundamental microcontroller concepts like timers, interrupts, basic input/output devices, analog-to-digital conversion, and event-driven programming. For example, a lab might involve periodically sampling the temperature sensor over the SPI bus using an interrupt and displaying the result on the LCD. To accomplish this, students work with high-level software abstractions for onboard peripherals like a keypad and LCD to develop their applications using the concepts explained in lecture. This allows students to learn the role of embedded systems in real-world applications and demonstrate fundamental concepts.

For students wishing to further their understanding of embedded systems, ECE3849 explores the design decisions behind an embedded system demonstrate how they can be utilized for complex, real-time applications by integrating hardware and software. ECE 3849 replaced an older embedded systems course called ECE 3803, which also discussed real-time constraints, but with a primary focus on hardware concepts like memory interfaces and bus timing analysis ("Undergraduate Courses," 2012). The laboratory assignments for ECE 3803 challenged students to build a complete embedded system on a prototyping board, while developing software to demon-

strate a real-time embedded application. An example lab project for ECE 3803—a real-time digital oscilloscope—is shown in Figure 8.

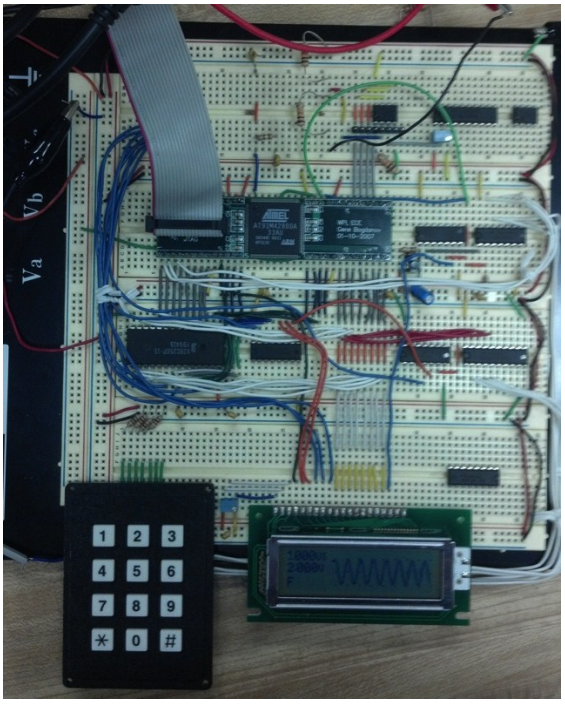



**Figure 8: Completed ECE 3803 Prototyping Board**

As demonstrated by the complex circuit in Figure 8, ECE 3803's labs required that students spend a great deal of time focusing on the hardware design concepts of building an embedded system from the ground up, which was the main focus of ECE 3803's course content. While these labs were often highly regarded by students, their focus on hardware neglects the advanced software design concepts necessary for modern embedded development.

In contrast, ECE3849 challenges students to develop a much more comprehensive understanding of software techniques to implement real-time systems while also learning how to integrate hardware devices into a complex system. ECE3849 emphasizes the concept of a real-time operating system (RTOS) in order to demonstrate complex real-time software concepts like scheduling, semaphore-based synchronization methods, and mailboxes. This allows students to demonstrate how the design and performance goals of an embedded application, which entails multiple tasks and shared data, can necessitate the use of an RTOS to facilitate the design goals. To facilitate learning of these concepts, ECE 3849 uses a pre-built development board to make peripherals readily available for development without the need for much wiring. In this way, the course facilitates RTOS-based labs that might otherwise be too complex for a student assignment.

Presently, ECE3849 teaches these wide-ranging concepts by demonstrating the challenges of interacting with hardware in real-time with and without an RTOS. To do this, the first two offerings of ECE3849 were developed around three main labs:

1. Develop a real-time digital oscilloscope using interrupt-driven programming (without an RTOS)
2. Port the oscilloscope implementation to an RTOS and add a spectrum analyzer mode
3. Develop a frequency meter for the RTOS-based implementation using timer data from another system connected via CAN bus

The oscilloscope lab, which comprised seven-weeks of development in ECE3803, is now completed in only two weeks, making room for the latter, more conceptually-rigorous labs, which challenge students use an RTOS as well as synchronization and networking while adding more real-time tasks to an already complex system. To support this, ECE3849 uses a development platform that allows students to implement systems while focusing on the software tasks involved in the design of a very complex multitasking system. The course currently uses a development board for Texas Instruments' (TI) line of Stellaris ARM microcontrollers, a TI LM3S8962 Evaluation Board, which is shown in Figure 9 running the digital oscilloscope for the first lab.

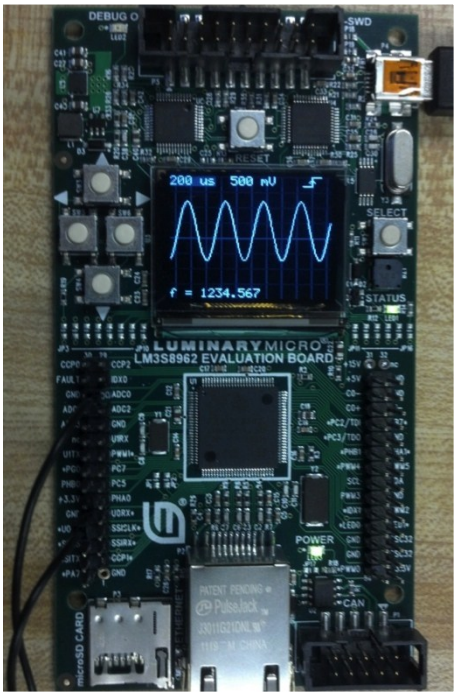



**Figure 9: Current ECE 3849 Development Board with Oscilloscope Lab**

While the current development board for ECE3849 allows for some interesting labs, it includes a rather limited set of peripherals for students to allow for lab designs. The development board includes the following devices (Texas Instruments, 2010):

- **User input/output**: 5 buttons, 128x96 monochrome display
- **Network**: CAN bus, Ethernet controller
- **Storage:** SD card slot

This set of peripheral devices is rather limited: it offers only a few hardware peripherals that can support labs without additional hardware to connect to the board. Since the interface between hardware devices and software is a core component of ECE 3849, instructors could develop more comprehensive labs with a platform that provides more hardware interaction. Thus, the ECE department wants to investigate a more capable hardware platform as a means of facilitating student learning in the course.

## 2.4. How are students at WPI frustrated by embedded systems design?

As discussed in section 2.2, students are often frustrated by the course material due to the disparity between their background CS and ECE courses and the complex hardware and software concepts required for embedded systems design. In the first two offerings of ECE 3849, students found it difficult to implement their labs due to the magnitude and complexity of software development required for their assignments. When compared to students' background courses like ECE 2049, CS1101, and CS2301, the software implementation for ECE 3849's labs introduce students to the following challenges:

- **Large Codebase and Software Libraries**: TI provides a software library for their Stellaris microcontrollers to provide an API for interfacing with the microcontroller's registers and the peripherals on their development boards. In addition, TI has provided drivers for the peripherals on the development board like the display. These libraries are used throughout ECE 3849's labs as they implement many common tasks for interacting with the on-chip peripherals, meaning that students' labs must use the libraries and drivers in their code. To do this, students need to understand how to read and utilize the API documentation as well as integrate the preexisting codebase into their software environment. In ECE 2049 and CS 2301, students' code typically comprises only a single file and does not use many (if any) external libraries or documentation, so this is a very new concept for students.
- **Extensive Debugging**: Writing software for an embedded system requires that students are heavily reliant on the debugger. Unlike general purpose computing, embedded systems do not often support **printf()** or even have a console to examine their program's output. In addition, debugging for an embedded system with multitasking introduces a new set of challenges for students, including understanding debug techniques for multiple tasks with real-time constraints and hardware devices. In CS 2301, students are introduced to the debugger, but are not required to rely on it this extensively. Students begin to learn these techniques in ECE 2049, but are not required to apply their knowledge to complex codebases or multitasked systems.
- **Advanced IDE Features**: Students in ECE 3849 use TI's Code Composer Studio (CCS) IDE. CCS is based on Eclipse, a popular open source IDE for Java and C which students learn to use in CS 2301. However, ECE 3849 requires that students manage a large

codebase in multiple code projects and understand how to extensively use the debugger. In addition to understanding the fundamental principles of a build environment and a debugger, students also need to learn how to apply these concepts to the software they use in lab. Given that CCS (and Eclipse) supports a vast array of features, this is a significant challenge for students. While CS 2301 begins to teach students how to use Eclipse, it does not have time to teach students the intricate complexities of using its advanced features and managing a large codebase.

In the first two offerings of ECE 3849, students found it difficult to make progress on their labs due to their lack of preparation in these concepts. For example, students cited that they required a significant amount of time in lab just to set up their build environment or understand a build error before they could proceed with their lab assignments. While these concepts are incidental to the embedded systems concepts in ECE 3849, they are critical to students' understanding of the software engineering concepts inherent to embedded systems design. Since students do not have the opportunity to learn these concepts in their other courses, ECE 3849 needs to find ways to integrate students' existing backgrounds into the course and teach students the more advanced software engineering concepts in coordination with the embedded systems material. Therefore, ECE 3849 and other embedded systems course need to take a comprehensive approach to teaching students the hardware and software concepts inherent embedded systems design.

## 2.5. Chapter Summary

Embedded system design requires the integration of both hardware and software engineering techniques to develop complex hardware-software systems. Therefore, a major concern of teaching embedded systems is ensuring that students are fluent in these hardware and software concepts as well as the domain-specific knowledge of embedded systems design. At WPI, students learn fundamental hardware and software design concepts in their background courses, but the material covered is insufficient for teaching students how to design modern, advanced embedded systems.

In ECE 3849, students learn to develop advanced embedded systems with real-time constraints using hardware design concepts like peripheral devices as well as software design concepts like multitasking and real-time operating systems. To do this, ECE 3849 uses a pre-built development board with a few onboard devices and challenges students to develop software for embedded systems in labs. While the current system provides a starting point for teaching students how to design embedded systems, a more comprehensive approach is needed to facilitate more comprehensive labs on a hardware level and help alleviate students' frustrations with the software design and engineering portions of their labs. In order to help alleviate these challenges, my design must provide a comprehensive teaching platform to help ECE instructors teach

students how to develop advanced embedded systems that integrate hardware and software engineering.

# 3. Development Board Design Process

This section describes my design process for determining the architecture of this development board. As described in the previous section, a development board enables experimentation with embedded systems design by providing a microcontroller and peripheral devices to support embedded applications. To design a development board that could best help alleviate the challenges students and instructors face in ECE 3849, I identified the following objectives:

1. Determine the development board's explicit and implicit requirements by interviewing key stakeholders and inferring from ECE 3849's course goals
2. Develop a system concept based on these requirements, including specifications that guided the selection of components for the design
3. Refine this system concept into a concrete design by selecting the key components: the microcontroller and peripheral devices
4. Implement the design by integrating the components on a PCB and demonstrating how the system can be used in lab

The following sections describe my design process for carrying out the objectives listed above.

## 3.1. Objective 1: Determine stakeholders' implicit and explicit requirements

The following stakeholders for were identified for my design:

- ECE Department Faculty
- Course Instructors teaching ECE 3849
- Students taking ECE 3849

### 3.1.1. The ECE Department

The computer engineering professors in the ECE department are the architects of ECE 3849. They hold the vision for what concepts the course should impart to students in order to provide useful design skills to use in a professional environment. In addition, the Department manages funding for ECE 3849 and its lab materials and would also be responsible for deploying it to students in labs. Therefore, the ECE department can provide insights regarding the kinds of applications the board should provide as well as cost and physical requirements for deploying it in lab.

### 3.1.2. ECE 3849's Course Instructor(s)

Professor Gene Bogdanov was responsible for teaching the first two offerings of ECE 3849 in B and D terms of the 2012-2013 academic year. Before the course was first offered, he select-

ed the current development board for the course and designed the three labs that currently make up the course, which implement a digital oscilloscope with frequency meter. In doing so, he also devised all of the lecture material for the course, determining how students are presented the very diverse range of concepts presented in the course.

### 3.1.3. Students taking ECE 3849

As students in the course have used the current development platform and implemented the existing labs, they can explain what aspects of the labs they found challenging or provide recommendations for new, possibly more exciting labs. In this way, students in ECE 3849 could provide useful recommendations for how my design could help make embedded systems design more accessible to students in future offerings of the course.

In order to assess the needs of these stakeholders, I conducted informal interviews with each party at various stages of my design. During the initial phases of my project, these interviews included general questions to help identify the capabilities my design should provide for students and instructors, including the following:

- Ideally, what would you like to be able to accomplish in ECE 3849 labs?
- What embedded systems concepts should be stressed in labs?
- Can you name an exciting project that you would like to see in lab?

In addition, these interviews helped to get a sense of the overall requirements for my design in terms of usability, deployability, and cost, which shaped how I devised the overall concept for my system. These requirements are explained in detail in section 4.

## 3.2. Objective 2: Synthesize requirements to develop system concept

Once I had developed a set of requirements from each stakeholder described above, I needed to determine how these requirements contributed to the design of a complete system. Based on the requirements I gathered from the instructors and students, I identified a set of engaging applications that my development platform could support. This resulted in a list of candidate peripheral devices that my design could provide to support the applications.

I then mapped these applications to the specific course challenges of ECE 3849 described in section 2 to discern how my design could effectively contribute to the course goals as well as engage students in embedded systems design. This justified how my design could help meet the course goals and the specific challenges students face in lab. In addition, I refined the requirements provided by instructors and students to a set of specifications that dictated how my design could be feasibly implemented as a lab platform for ECE 3849. From there I could begin the process of selecting discrete hardware components for my design, which is detailed as part of section 3.3.

## 3.3. Objective 3: Select key design components

Once I had developed a set of specifications to guide my implementation and a set of components that my design should include in order to provide a capable design for the course, I could begin selecting hardware components for the design. Fundamentally, a development board contains two main components: a microcontroller, and a set of peripheral devices with which it talks to the outside world. This phase involved selecting all of the peripheral devices described in the previous phase as well as a suitable microcontroller to support them.

My primary design challenge in this phase was selecting a microcontroller and set of peripherals that could meet my design requirements while proving feasible to implement in this project's timeframe. This involved a trade-off analysis of various microcontroller board implementations, including pre-made boards as well as the design of a custom microcontroller board, which could support peripheral devices for my design. This was a highly iterative process that involved reconsidering the candidate peripherals and initial requirements in order to balance the board's capability with feasibility of implementation. During this phase of development, communication with the stakeholders was maintained in order to ensure that new designs met the defined requirements and could be feasibly deployed in the laboratory. The result of this phase of development was a complete architectural picture of the board's hardware, including a microcontroller and peripherals.

## 3.4. Objective 4: Implement the design and demonstrate functionality

Once the hardware components to include on the development board had been selected, I needed to integrate them onto a PCB to develop a prototype to demonstrate how the system could be deployed to students in lab. This phase involved the creation of a PCB layout to integrate the selected components. In addition to ensuring functionality of the microcontroller and its peripherals, the PCB design was heavily influenced by the stakeholders' requirements such that the physical hardware provided to students in lab provided an engaging and user-friendly platform for labs.

Once the PCB layout was completed, the components were extensively tested to verify their functionality when integrated into the overall system. This included development of demonstration code for the selected microcontroller and each peripheral on the board to show how each component could be used in a complete system.

## 3.5. Chapter Summary

My development board design required identifying the requirements of ECE instructors and students, using them to define a system concept, and determining the components of the system to realize a final design. This was a iterative process: as discovered the implementation details of the candidate microcontrollers and peripherals, my system architecture evolved to identify a feasible design that could best fit ECE instructors' and students' requirements. Much of this

process involved studying the tradeoffs involved with certain implementations with respect to instructors' and students' requirements. The next section details my investigation of potential implementations for my development board and my design process for realizing the final system architecture.

# 4. Development Board System Architecture

This section describes the how the development board was designed and implemented following the methods discussed in section 3. It describes the initial requirements given by the three key stakeholders of my design: the ECE department, the current instructor of ECE 3849, as well as students taking the course. It then goes into detail regarding how these requirements were mapped to specifications based on the course goals and describes the design decisions involved in defining the system architecture for this design.

## 4.1. Identifying stakeholder requirements

The initial requirements for the development board were based on informal interviews with three key stakeholders: the ECE department, which holds the general vision for the course; the current instructor of ECE 3849, Professor Bogdanov; as well as students taking the course. The general requirements expressed by the stakeholders are described below.

### 4.1.1. The ECE Department

Professor Looft described that a development board capable of supporting ECE 3849's labs should:

- **Allow for a variety of labs that fit ECE 3849's course material:** The current development board does not provide many options for laboratory assignments, as it is limited to only a few peripherals. A more capable board should allow instructors to design a variety of labs that challenge students to develop real-time systems with a single set of hardware.
- **Support "exciting" lab assignments that demonstrate engaging real-time applications**: The digital oscilloscope lab supported by the current development board provides an interesting laboratory project for ECE students. However, a more capable development board could support real-time labs with more demonstrable and exciting applications, such as navigation or motion tracking. This could provide more exciting labs for ECE students as well as RBE students.
- **Allow for feasible deployment to the 25 lab benches in AK113**: Just like the current lab platform, students must be able to use my design in the teaching laboratory to complete their assignments. An approximate cost target was set at $500 per development platform.

### 4.1.2. Current ECE 3849 Instructor

Professor Bogdanov outlined some additional requirements to allow the development board to be feasible in the course he designed, noting that an implementation useful for instructors should:

- **Adhere to the existing course design**: Professor Bogdanov has already developed a rigorous set of lectures and labs for the first two offerings of the course. A new development board should require minimal alteration of the existing course material to avoid requiring the existing course staff (including Professor Bogdanov and current tutors and TAs for the course) to relearn the subtle implementation details of a new hardware platform.
- **Enables labs feasible for students to implement in the course's timeframe**

### 4.1.3. Students taking ECE 3849

Students taking the course noted that a development board should:

- **Allow for labs applicable outside of the course:**
- **Is user-friendly enough to allow students to complete their labs**

## 4.2. Developing system concepts

The stakeholder analysis conducted in the previous section provided a set of general requirements for the development board. I defined a concrete list of specifications that guided my overall design based on these requirements and the course concerns outlined in section 2. In this way, I could target my design to instructors' and students needs as well as the educational objectives of the course.

In order to develop an architectural picture of the system, I needed to identify what types of components to include on the development board to fit the course requirements. Fundamentally, a development board for ECE 3849 contains three major components: a microcontroller for students to program in their labs, user input and output devices for students to control the system, and peripheral devices like sensors to provide input data for computations, as depicted in Figure 10.

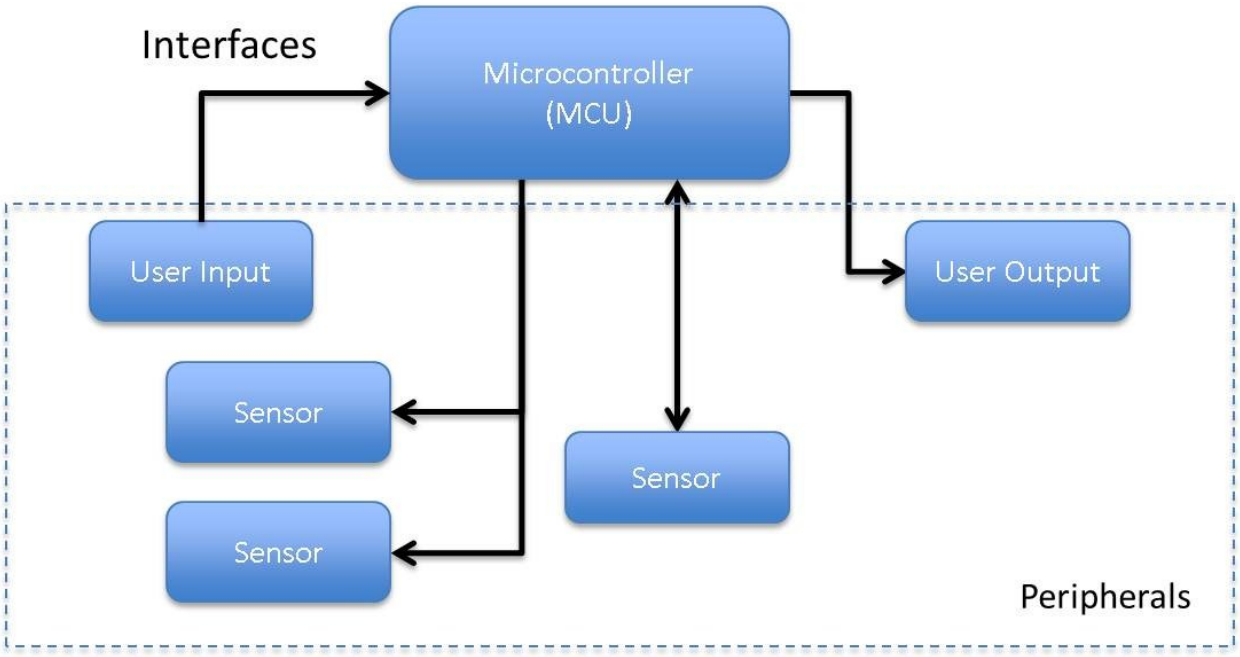



Figure 10: Fundamental Embedded System Concept

Since ECE 3849 stresses both hardware and software concepts as part of its labs, the microcontroller, which allows students to demonstrate software concepts, and peripheral devices, which allows students to implement hardware concepts, must both be evaluated against the stakeholder and course requirements. In addition, the interfaces used to connect the peripherals to the microcontroller are important course topics in ECE 3849, meaning that the interfaces for the peripherals are critical to the design as well. The design requirements and the course goals were used to guide selection of these components and interfaces.

### 4.2.1. Identifying requirements for peripheral devices

In an embedded system, peripheral devices define how a microcontroller communicates with the outside world. Peripherals provide a very wide array of functions, including sensors that provide data for product functions, input or output devices like buttons or displays that create a user interface, and network interfaces that enable communication with other devices. For this design, it follows that the peripherals included on the development board need to enable students to experiment with embedded systems in their labs. Despite the immense variety of peripherals available, these devices could be evaluated based on their contribution to the overall system on three main criteria:

- **Functionality**: Any peripheral device affords the development platform some function, either by providing data (like a button or sensor) or by using data in a useful way (like a display). The functionality afforded by each peripheral device was evaluated against the

specific requirements of this development board, like its relevance to the course goals and feasibility of implementation on a development board.

- **Interface**: By definition, peripheral devices must connect to a host device—in this case, a microcontroller—which operates it in some way. Microcontrollers provide a variety of these interfaces, each of which has different speed and implementation challenges. Part of ECE3849 (and ECE2049) involves the study of the tradeoffs associated with selecting peripheral interfaces. Therefore, peripherals were evaluated based on the feasibility of using its interface on the development board in lab and their contribution to students' knowledge of peripheral interfaces.
- **Cost**: In addition to monetary cost, all peripheral devices have implementation costs, including the computational and hardware resources required to utilize a particular device. These costs were evaluated based on the feasibility of adding the device to my hardware design as well as the software implementation of the class of embedded microcontrollers available for this type of embedded system.

#### *4.2.1.1. Peripheral device requirements*

Based on the stakeholder requirements and the course goals, we identified that the peripherals selected for my design should:

- **Utilize a variety of buses and interfaces common to embedded systems programming to communicate with the above peripherals**: As outlined in the course aims analysis in the previous section, one of the main course goals is to teach students about peripheral interfaces and their uses in embedded applications. Given this, diversity among available interfaces is a key component of the design.
- **Include a variety of engaging devices common to other embedded applications:** Since instructors and students desired a system that is applicable to real-world embedded applications, my design should include peripheral devices that students are likely to see in a professional environment. In doing so, these peripherals can include devices that students are likely to use in their other ECE or RBE courses. This could create a connection to students' background knowledge, making the task of programming the system more engaging.
- **Withstand prolonged student use and potentially abuse:** Based on the ECE department's experience with the development boards for ECE 2049, instructors noted that the devices for my development board should withstand intense student usage in lab.

Based on this, my design needed to provide an assortment of devices and interfaces to allow instructors the flexibility to create new lab assignments. This influenced how I selected the number and type of peripherals for my design.

The reliability requirement also introduced an additional design challenge: in order to ensure that my design was resistant to device failures due to student use or abuse in lab, we decided to select only peripheral devices available as pre-built modules that would require minimal hardware on my PCB to implement. These pre-built modules, commonly available from ECE hobbyist retailers like Sparkfun, consist of small PCBs, often called "breakout boards," that include a particular peripheral device and the incidental components needed to support it. An example of such a breakout board is shown in Figure 11, which includes an accelerometer (Sparkfun, 2013).


**Figure 11: Example Sparkfun Breakout Board**

By relegating most of the necessary components to breakout boards, my development board needs to only provide the appropriate power and data connections to use the peripheral, which connect to the breakout board via header connections. Thus, when a component in the design fails, the module can be replaced to restore functionality.

#### *4.2.1.2. Selecting Candidate Peripherals*

Using the requirements developed in the previous section, a list of candidate peripheral devices was selected to demonstrate common tasks in real-time systems that students are likely to encounter in other course projects and in their professional experience. Since ECE 3849 is catered to both ECE and RBE students, peripherals commonly used in both disciplines were selected, enabling students to relate their labs directly to their other courses.

The initial list of candidate peripherals to include in my design is shown in Table 1.

Table 1: List of Candidate Peripherals

| Device | Applies to | Course aim | Comment |
|---|---|---|---|
| **Keypad** | All | User input | This type of basic I/O is necessary for user input for any application and also provides an example of simple I/O interfaces |
| **Character Display** | All | User output | |
| **Graphics Display** | All | User output; Real-time applications | A lengthy task like updating a large, graphical display could serve as one of many tasks in a real-time lab |
| **External Memory** | ECE | Interfacing with memory devices | Memory interfaces are a common topic in embedded and microprocessor design. This could include volatile memory devices like SRAM or DRAM or nonvolatile devices like flash or EEPROM. |
| **USB** | All | Interfacing with peripherals | Could allow for labs using any real-world USB peripherals |
| **DAC (Digital-to-Analog)** | ECE | Real-time applications | Audio processing with on-chip ADC and external DAC could create a real-time lab |
| **Audio Codec** | ECE | Real-time applications | |
| **Temperature sensor** | ECE | Interfacing with peripherals | Various sensors could provide useful examples for peripheral labs |
| **Light Sensor** | ECE/RBE | Interfacing with peripherals | |
| **Accelerometer** | RBE | Real-time applications | Could be utilized in a lab involving navigation |
| **IMU** | RBE | Real-time applications | |

This initial list of peripherals was created in collaboration with ECE instructors to help ensure that these devices could provide a variety of labs feasible for the course. As my design progressed to evaluate available microcontrollers, this list was refined based on the feasibility of integrating these devices with a microcontroller in my design. From there, I was able to analyze the available peripheral devices using the above criteria for integration into my design.

#### *4.2.1.3. Identifying microcontroller requirements*

Based on further discussion with the stakeholders regarding the course goals, we identified the following requirements to guide selection of the microcontroller used for my development board should support:

- **Software development using an RTOS and interrupt-driven programming:** ECE3849 inherently entails programming with and without an RTOS to demonstrate both types of real-time software development, meaning that this development platform should do the same.
- **USB debugging:** ECE instructors noted that they encountered difficulty maintaining a lab environment that could support the JTAG interface for programming the development boards for ECE 2049. As such, they specifically requested that my design connect to the lab computers using USB, meaning that it must include the necessary hardware for a USB debugging interface.
- **Programming in C using a robust IDE like Eclipse:** Both ECE 2049 and ECE 3849 utilize C programming to complete their labs, as defined by the course goals, meaning that my board should do the same. In addition, the current offerings of ECE 3849 introduce students to TI's Code Composer Studio; a robust IDE based on Eclipse that students are likely to see in industry. My design should also use robust development tools like these to ensure that students are prepared for the software challenges associated with modern embedded systems programming.
- **Compatibility with ECE 3849's existing course material:** Professor Bogdanov requested that a new development board maintain as much compatibility with his existing course design as possible
- **An assortment of I/O buses and interfaces to support the candidate peripheral devices:** As discussed in the previous section, my design necessitates the integration of a number of peripheral devices.

These hardware and software support requirements dictated the selection of a microcontroller for my design. In order to maintain compatibility with ECE 3849's existing course material, we decided to limit the microcontrollers evaluated for my design to the TI Stellaris ARM family, the same family of microcontrollers as used on the current implementation. In this way, most of the course material, which includes architectural descriptions of the microcontroller and specific lectures dedicated to using TI's SYS/BIOS RTOS, could remain unchanged. In addition, this ensures that the same development environment (and programming language) could be utilized, therefore requiring minimal changes to the existing lab setup in AK 113.

The I/O requirement introduced an additional design challenge: the microcontroller selected for my design needed to support the peripheral devices selected in the previous section. In doing so, the microcontroller needed to support the variety of interfaces discussed in ECE 3849, including:

- **GPIO (General Purpose Input/Output):** This is the simplest interface for controlling input and output to a microcontroller. A developer can simply designate available GPIO ports as input or output and utilize them to directly read or write bits. This allows a very simplistic yet powerful degree of control by simply defining whether a logical zero or one

should be the output on a particular pin, according to the application. GPIO is frequently used to implement control signals like chip selects, serial or parallel bus interfaces, or generate digital waveforms or signals for other uses.
GPIO pins on a microcontroller are typically divided into ports, or groups of 8 or more pins on a single register that can be read or written simultaneously. These contiguous GPIO pins can be utilized for implementing simple parallel interfaces without many read-write operations.

- **Serial Peripheral Interface (SPI):** SPI provides a four-wire bus interface to many peripheral devices; it uses four signals—a clock signal, a chip select, and one signal for data in each direction. One device on a SPI bus serves as a master, while the other devices act as slaves and wait for signals from the master. The master device provides a clock signal and selects a device using its dedicated chip select signal; after this, the master and slave can transmit and receive data simultaneously using the two data lines. The clock signal can be as fast as the master and slave support, allowing for arbitrarily fast or slow communications between devices.
- **$I^2C$ (Inter-Integrated Circuit):** $I^2C$ provides a two-wire bus interface to a wide range of peripheral devices, though in a more rigidly defined manner than SPI. Every $I^2C$ device is assigned unique a 7-bit address during production, which is used to access each device on the bus. This eliminates the need to allocate a specific chip select line for each device as for SPI. $I^2C$ utilizes two signals, one for data and the other to provide a clock. The specification defines rigid protocols for the master and slave nodes to use these two lines to address each other, start a transfer, acknowledge each transmitted byte, and even provide basic flow control. As the standard evolves, $I^2C$ specifications have defined a number of speed modes, ranging from 100kbps to 5Mbps. This development board supports the original 100kbps standard and 400kbps "fast mode" (Phillips Semiconductors, 2000; TI, 2010b).
- **PWM (Pulse-Width Modulation) Generators:** PWM signals control power to devices by transmitting a square wave with a varying duty cycle: the larger the duty cycle (i.e. the more time the output line stays high), the more power is transferred to the output device. This is used for controlling devices like motors or LEDs, which frequently require a variable intensity.
- **Analog Inputs:** Most microcontrollers provide an on-chip analog-to-digital converter (ADC). These inputs can be passed directly to the microcontroller's ADC for use with any analog peripherals.

By including these interfaces, my design would make it possible for instructors to create labs that teach students how to implement these hardware peripherals. In order to allow instructors to create a variety of hardware-based labs, we decided that my design should utilize peripherals on at each of these interfaces or provide headers on the PCB for them to allow for future expansion.

## 4.3. Refine system concept to system architecture

The system concepts defined above provided a general outline for designing the overall system. From there, I needed to refine my design by selecting components for the peripherals and microcontroller. This phase of the design was an iterative process: since the peripherals in the system are dependent on the capabilities and interfaces on the microcontroller, my peripheral selections were affected by my choice of microcontroller.

### 4.3.1. Selecting the Microcontroller

As discussed in section 4.2.1.3, we decided to select use microcontrollers in the TI Stellaris ARM family in order to maintain compatibility with ECE 3849's existing course materials. I examined the following microcontroller options, including include pre-built microcontroller development boards and a custom made solution; these options are discussed in the subsections below:

- The current development board for ECE 3849: the Stellaris LM3S8962
- Other Stellaris Cortex-M3 development boards
- A custom-designed microcontroller board
- TI's Stellaris LM4F232 development board

#### *4.3.1.1. Current ECE 3849 Development Board: Stellaris LM3S8962*

An ideal choice for a microcontroller board that maintained compatibility with ECE 3849's existing course material would be the same development board used for its current labs, the Stellaris LM3S8962, shown with its components labeled in Figure 12 (TI, 2010b).

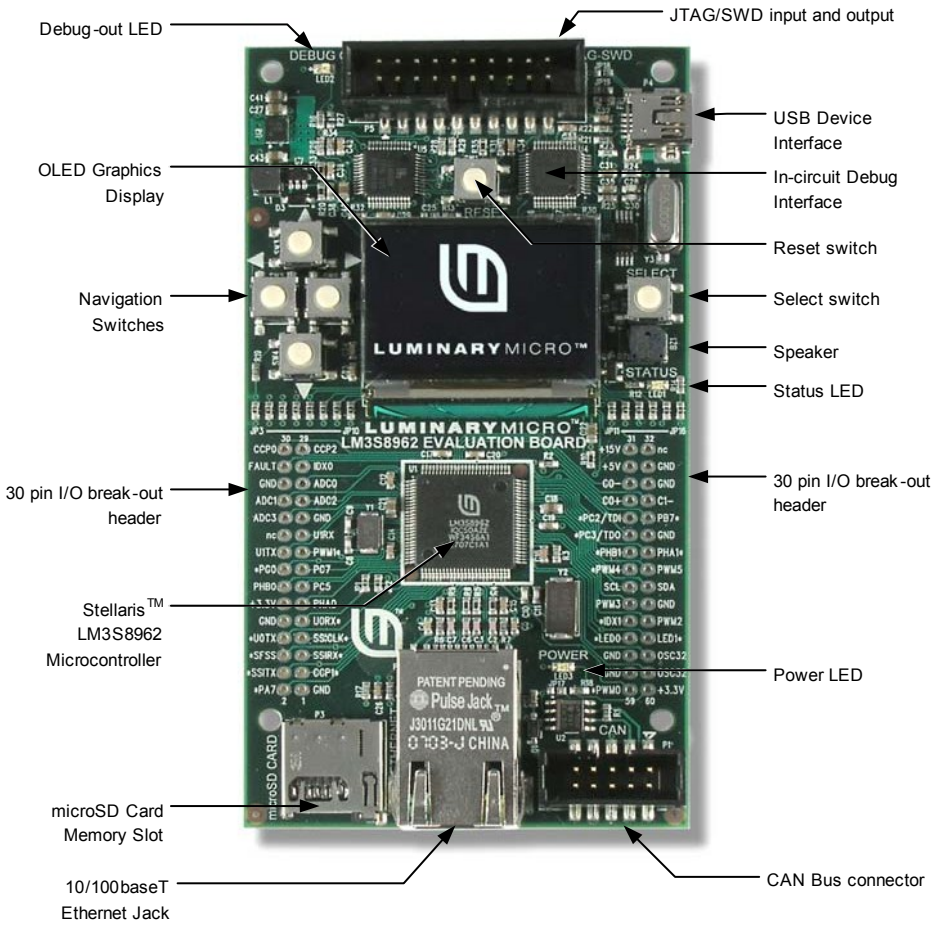



**Figure 12: Labeled Stellaris LM3S8962 Development Board**

This development board already provides a few useful peripheral devices, including a 128x96 monochrome display, 5 buttons, an SD card slot, an Ethernet jack, and a CAN bus controller. Any extensions on this board would need to take place using the two 30 pin expansion headers on the board. However, many of the 60 pins on the expansion headers provide connections to either the power rails or the onboard devices, which leaves relatively little room for expansion. In many cases, the pins available on the headers are shared, or multiplexed, with onboard devices like the buttons. After eliminating these shared connections, the development board leaves only the following interfaces for expansion:

- 11 GPIO pins on non-contiguous interfaces, shared with the signals listed below
- 1 SPI bus, shared with the display and SD card slot
- 1 $I^2C$ bus
- 1 UART
- 4 PWM Outputs
- 1 Comparator
- 2 Analog inputs

Most notably, the 11 non-contiguous GPIO inputs do not provide a group of 4 or 8 GPIO pins that would enable a parallel interface for use with peripherals with high data rates like a graphical display. Creating such a bus with non-contiguous inputs would involve individually writing each bit in a separate instruction. This method would not only be cumbersome to use and sacrifice performance, but not be representative of the types of parallel interfaces available on devices in a professional environment. As such, this development board is not a good candidate for demonstrating parallel interfaces, as it would make implementation of the keypad and displays difficult.

Moreover, the SPI bus available on the board is already used by the onboard display. As a display represents a low-priority, high-bandwidth task in an embedded system, this could potentially limit the data rate for any other devices connected to the same bus. Based on this, the SPI bus on the device would be difficult to use with higher-priority tasks like sensors. As this would severely limit the number of peripherals and interfaces available for my design to demonstrate or complicate the design such that it would be opaque to students, we decided to pursue other microcontroller options.

#### *4.3.1.2. Custom microcontroller board design*

Since the Stellaris LM3S8962 development board provides very limited I/O capabilities, I also investigated the feasibility of developing a custom PCB containing a microcontroller with enough I/O to support the candidate peripherals. TI provides hundreds of models of Stellaris microcontrollers. To identify a possible configuration with sufficient I/O peripherals, the Stellaris LM3S9B96, was selected as an example for further investigation. As one of the largest microcontrollers available using the ARM Cortex-M3 core, as used by the LM3S8962, a design using the LM3S9B96 could provide enough I/O for the candidate peripheral devices, while maintaining compatibility with most of the existing course material.

The LM3S9B96 is available in a 100 LQFP package, meaning that is has 100 available pins distributed a variety of interfaces. Figure 13 shows the microcontroller's internal block diagram (Texas Instruments, 2010a, p. 56).

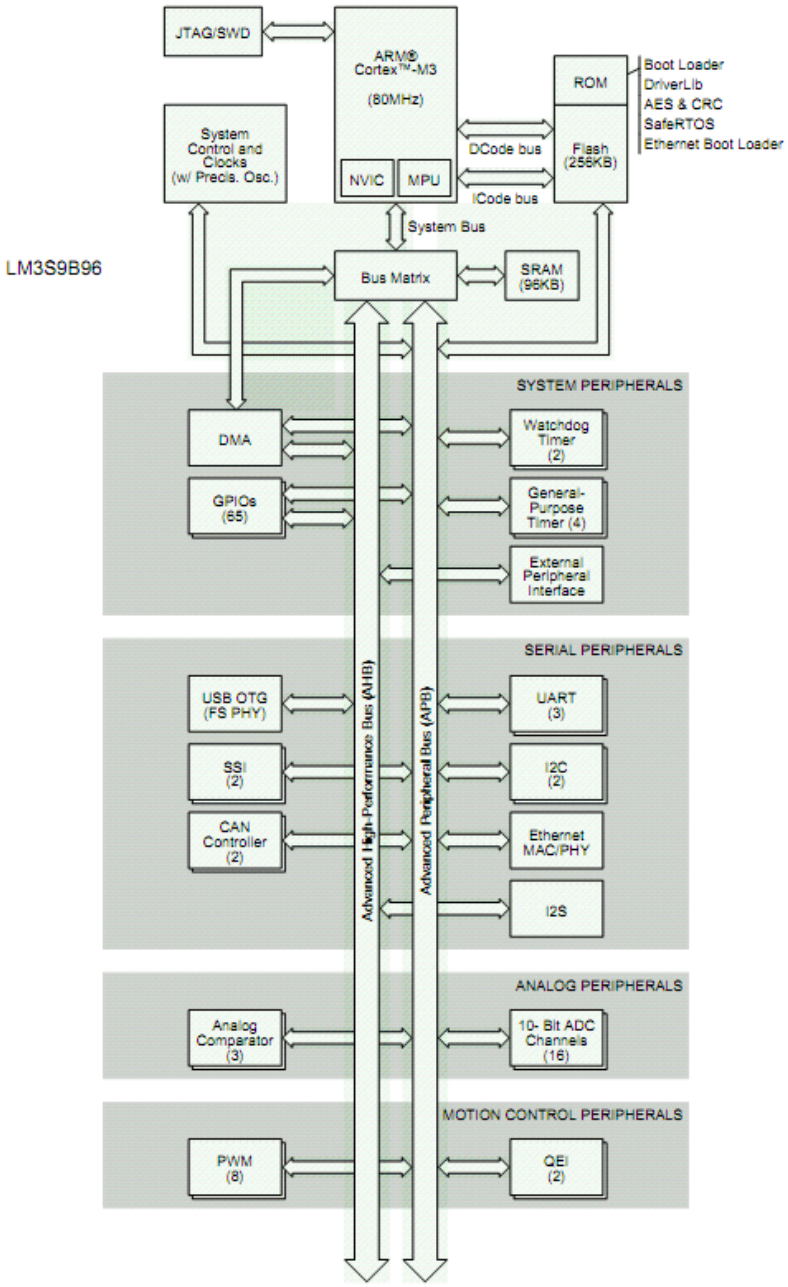



**Figure 13: LM3S9B96 Block Diagram**

Most notably, the LM3S9B96 contains the following notable interfaces:

- 2 SPI buses
- 2 $I^2C$ buses
- 3 UARTs
- 16 Analog inputs
- Ethernet controller
- External Peripheral Interface for implementing memory devices

- USB controller
- 3 Comparators
- 2 CAN controllers
- 8 PWM Inputs
- Up to 65 GPIO pins, multiplexed with the above signals

Based on this, the LM3S9B96 has enough available interfaces for the above peripherals, as it would allow the high data-rate devices to be distributed across multiple SPI and $I^2C$ buses to ensure that the buses would not become congested. Notably, this microcontroller also contains a memory bus interface called the External Peripheral Interface that allows for a custom implementation of memory-mapped I/O ("Stellaris LM3S9B96", 2012, p. 60). This could allow for the implementation of a complex memory device (like DRAM or SRAM, which are covered in the lectures) on my design, which is not possible with the current development board or many other microcontrollers, which do not often provide access to the memory bus.

While this custom-designed approach would provide the maximum capability in a microcontroller for my design, we deemed it infeasible to implement due to the time and resource constraints of this project as well as the assembly requirements of the laboratory setup. A critical setback of this implementation would be the need to design the circuit and PCB layout for the microcontroller. Since it the microcontroller operates at a maximum clock rate of 80 MHz, the layout process would be very complex, as signals over 10 MHz can introduce substantial RF interference (Guy, 2013).

In addition, the microcontroller design would also have required a solution to support USB debugging. Development boards commonly implement this using small a microcontroller to convert JTAG signals to USB, or using an FTDI USB to Serial converter (TI, 2010b; Future Technology Devices International, 2012). However, the procedure for implementing this circuit is not well documented and appears to vary from microcontroller to microcontroller and relies heavily on drivers on the host computer to implement the interface. Furthermore, the current lab setup uses Code Composer's free license, which is available for Stellaris evaluation boards, but not custom microcontroller solutions (TI, 2013). Thus, a custom implementation would likely require that each lab station be given a full license for CCS, which was outside this project's budget.

Furthermore, all of the Stellaris microcontrollers examined were only available in fine-pitch physical packages, like the 100LQFP. When conducting initial testing with this microcontroller, it proved very difficult to manually solder to a breakout board. Upon further discussion with Prof. Looft, I learned that assembly for my design would be conducted manually by department staff and student workers. This means that assembling a design using one of these packages would be similarly difficult for a full-scale deployment, potentially leading to an unacceptably-high device failure rate. While the design could be assembled by a third party that could auto-

mate the process similar to large-scale circuit designs, the cost for assembly of a small quantity of microcontroller boards would be prohibitively high.

These challenges were identified by conducting limited prototyping with a LM3S9B96 microcontroller on a breakout board to determine the feasibility of soldering the microcontroller in-house and implementing a debug interface. However, the prototyping approach showed many additional costs, and potentials for unreliable or failure-prone designs, making it an unviable candidate for the cost, reliability, and usability constraints of a student lab platform.

#### *4.3.1.3. Other Stellaris development boards*

The Stellaris line of microcontrollers is relatively new, meaning that few microcontroller boards using them are available on the market. A few notable options were other evaluation boards provided by TI, including its popular Stellaris Launchpad, as well as an option from MikroElektronika called Mikromedia, a company which produces development boards for a number of microcontrollers, which included a large display. The Stellaris Launchpad and the Mikromedia boards are shown in (TI, 2012; Mikroelektronika, 2013).

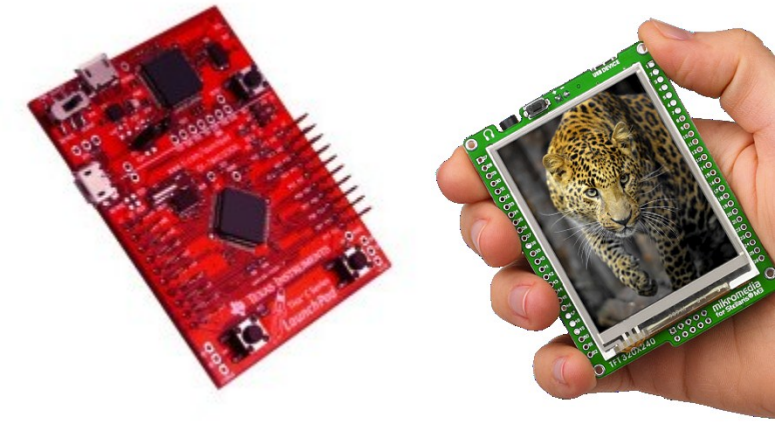


**Figure 14: TI Stellaris Launchpad and Mikromedia Development Boards**

These development boards demonstrated the same concern as the current LM3S8962 development board: insufficient I/O to support the candidate peripheral devices. Both devices did not provide enough SPI and $I^2C$ buses to allow for devices to be distributed across the interfaces to avoid congestion. In addition, both devices lacked a fully-available GPIO port, which would make any parallel bus implementations for the displays and keypad very complex.

While the Mikromedia board has the advantage of providing a very capable display, it also introduced concerns regarding the development environment required to use it. Mikroelekrtonika produces a custom toolchain, hardware debugger, bootloader and IDE to program their devices. It was unclear the extent to which the debug interface they produced would interface with Code Composer and the existing lab setup. Since a requirement of my design was to maintain

compatibility with the existing development tools and software libraries, this was not an acceptable alternative for my design.

#### 4.3.1.4. New Stellaris Development board: LM4F232

Midway through this project, , TI made available a development board using their new Cortex-M4 family of Stellaris microcontrollers, the Stellaris LM4F232. The LM4F232 evaluation board uses a 144-pin package Stellaris LM4F232H5QD microcontroller, which has enough I/O to provide on board peripherals to demonstrate the device's analog measurement capabilities while also allowing for expansion. Figure 15 shows the development board with its onboard components, including a small color display, a USB port, SD card slot, and a number of analog sensors (TI, 2012b).

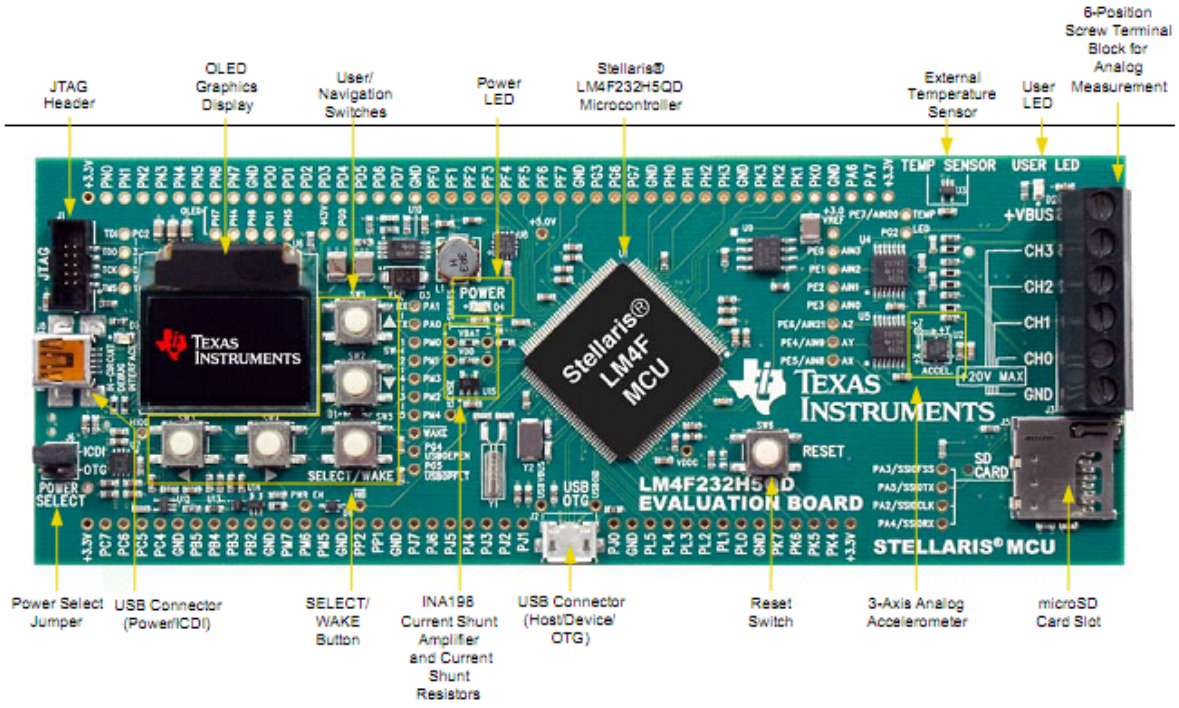



Figure 15: LM4F232 Development Board with labeled components

Most importantly, the LM4F232 contains 85 expansion pins on its expansion headers that are not shared with the onboard devices. The unshared peripherals include:

- 2 SPI controllers
- 6 $I^2C$ controllers
- 2 CAN controllers
- 6 UARTs
- 12 Analog inputs
- 16 PWM outputs

- 70 GPIO pins multiplexed with the above interfaces, including 3 contiguous ports

Based on this, the LM4F232 board provides enough unutilized peripheral interfaces to support the candidate peripheral devices. This includes the contiguous GPIO ports useful for implementing parallel devices.

A notable tradeoff is the change in microcontroller family from ECE 3849's current implementation, which uses an ARM Cortex-M3 core. This means that the core architecture differs slightly from the material in lecture. Upon a brief inspection of the key features of both cores, it appears that the Cortex-M4's architectural differences from the Cortex-M3 include the addition of a hardware floating-point unit (FPU) and a DMA controller, therefore providing additional capabilities for the course while maintaining the same architecture (TI, 2012c). The Cortex-M4 core for this microcontroller also lacks an Ethernet controller, which is available on the LM3S8962, and the External Peripheral Interface, which would have allowed implementation of memory devices. While this is a loss in physical capability compared to the current implementation, neither Ethernet or memory devices are part of ECE 3849's current labs, meaning that the existing course material or lab designs will not be affected.

TI also notes that both families use the same driver libraries and IDE (TI, 2012b). This means that, despite the change in microcontroller family, most of the course material can be utilized with minimal updates, which we deemed an acceptable tradeoff for the net gain in capabilities.

#### *4.3.1.5. Microcontroller board summary*

The capabilities and implementation and monetary costs for each microcontroller implementation are summarized in Table 2.

Table 2: Microcontroller Board Analysis Summary

| Microcontroller Board Option | Type | Cost | Supports current labs | Supports existing tools | Expandability on SPI & $I^2C$ | Expandability on GPIO | Implementation Cost | Used for Design? |
|---|---|---|---|---|---|---|---|---|
| Stellaris LM3S8962 Evaluation Board | Cortex-M3 | $89 | Yes | Yes | Minimal, buses congested | No, no contiguous ports | Low | No |
| Custom Microcontroller Board based on LM3S9B96 | Cortex-M3 | Prohibitively high | No | Debugging with free license is unlikely | Multiple buses available | Yes, many available ports | High, requires new PCB design | No |
| Other Stellaris Evaluation Boards | Cortex-M3 | ~$12-100 | No | Yes | Minimal, buses congested | No, no contiguous ports | Low | No |
| Mikroelektronicka Board with Display | Cortex-M3 | $100 | No | Unlikely, custom debugger | Multiple buses available | No, no contiguous ports | Low | No |
| Stellaris LM4F232 Evaluation Board | Cortex-M4 | $150 | No | Yes, requires some lecture updates | Multiple buses available | Yes, many available ports | Low | Yes |

The available microcontroller options showed significant tradeoffs in implementation cost and capability for ECE 3849's labs. The Stellaris LM3S8962 development board used for the current offerings of ECE 3849 as well as the other Stellaris Cortex-M3 boards would not afford my design much functionality due to the lack of readily-available I/O interfaces for expansion. While the Mikroelectronika board or the custom solution would have allowed for more expandability, they did not fit the critical requirement of supporting the existing course material and development tools. Based on this, I selected the LM4F232 development board as the microcontroller implementation for my design, as it provided an acceptable balance of I/O to support the candidate peripherals device while allowing for compatibility with the existing lab environment.

### 4.3.2. Selecting the Peripherals

As discussed in section 4.2.1.2, we selected a list of candidate peripheral devices that could demonstrate a variety of embedded systems applications. The implementation-level design considerations for selecting components, based on the peripheral device requirements from the previous section, included:

- **Use easily removable components**: Since we decided that the development platform would use a modular design to allow for easy replacement of components, I limited my selection of peripheral devices to those available as pre-built breakout boards or components with through-hole packages. In this way, the components can be installed on the PCB using sockets to allow for replacement without soldering. This also reduces the implementation cost to assemble assembling the final design.
- **Utilize available interfaces provided by the microcontroller**
- **Use peripherals that can be supported by the microcontroller board's onboard power regulator circuitry:** The LM4F232 development board is powered by the same USB cable it uses for debugging, which provides a maximum current of 500mA when connected to a computer. According to the development board's datasheet, the development board can provide up to 260mA using the power connections its expansion headers, which gives the maximum current for use by the peripherals.

Table 3 summarizes my analysis of the peripherals selected for my design as well as some alternatives that were not selected, including justifications for my decision to implement each peripheral device. The full bill of materials for my design, which includes part numbers and ordering information for each component, is included in Appendix C.

**Table 3: Peripheral Device Selection Summary**

| Device | Selected Device | Cost | Interface | Justification | Included? |
|---|---|---|---|---|---|
| **Keypad** | Greyhill 3x4 Matrix Keypad | $14.20 | GPIO | • Commonly used in student projects<br>• Allows for simple GPIO lab example currently covered in ECE 3849's lectures | Yes |
| **Character Display** | Crystalfontz 16x2 Character Display | $19.48 | 8bit Parallel | • Demonstrative of simple parallel bus application<br>• Uses an industry-standard controller<br>• ASCII-based protocol is simple enough for a lab | Yes |
| **Graphics Display** | Crystalfontz 320x240 color display with touchscreen | $86.78 | 8bit parallel | • Provides a complex, low-priority task for real-time applications<br>• High resolution, color palette and touchscreen demonstrate exciting capabilities of modern embedded systems | Yes |
| **Touchscreen Controller** | TI ADS7843 Touchscreen controller | $11.36 | SPI | • Simple example of SPI device<br>• Adds an additional real-time task for a touchscreen-based lab<br>• However, not available as a breakout board or DIP package | No |
| **High-Speed Memory** | | | | • Allows for lab demonstrating memory interfaces<br>• No memory bus access available on selected microcontroller | No |
| **EEPROM** | MCP2764 16kbit EEPROM | $0.56 | $I^2C$ | • Simple application of $I^2C$ bus<br>• Provides a slow task for real-time labs<br>• Demonstrative of applications for low-speed, nonvolatile memory | Yes |
| **USB Output** | Sparkfun FTDI basic breakout | $14.95 | UART | • Extremely common peripheral device used on many student projects and professional embedded systems<br>• USB already provided in hardware by the development board, making this option redundant | No |
| **DAC (Digital-to-Analog)** | Sparkfun MCP4725 DAC breakout | $4.95 | $I^2C$ | • Commonly used in student projects<br>• Allows for real-time labs with on-chip ADC | Yes |
| **Audio Codec** | | | | • Would allow for more complex audio processing labs that demonstrate real-time operation<br>• Only commonly available using the $I^2S$ serial interface, which is not available on the selected microcontroller | No |
| **Temperature sensor** | Sparkfun TMP102 breakout board | $5.95 | $I^2C$ | • Commonly used in student projects<br>• Simple application of $I^2C$ bus | Yes |
| **Light Sensor** | Sparkfun TMET6000 breakout board | $4.95 | Analog | • Simple application of analog measurement<br>• Similar analog peripherals commonly used in student projects | Yes |
| **Accelerometer** | Sparkfun ADXL345 breakout board | $27.95 | SPI | • Extremely common peripheral device and breakout board common to many student projects, especially robotics applications<br>• Demonstrative of real-time applications using the SPI bus | Yes |
| **IMU** | Sparkfun MPU-6050 Gyro & Accelerometer | $39.95 | $I^2C$ | • Could be integrated with the ADXL345 in one real-time lab<br>• While this device also provides an accelerometer, it is sensible to provide an alternative to demonstrate implementation tradeoffs of using both peripherals on one chip | Yes |

The total cost for the selected peripheral devices was $205.89 at single quantity prices, leaving sufficient room in the $500 budget for the development board, PCB, and mounting hardware.

An analysis of the current draw of each peripheral device is shown in Table 4. This information was identified from each component's datasheet. For devices with varying standby and operating current, their current usage was conservatively estimated using the highest specified operating current to account for the maximum power usage by all peripherals during operation. Operating currents in the microamp range were conservatively rounded up to 1mA.

**Table 4: Selected Peripheral Current Usage**

| Peripheral Device | Current Usage |
|---|---|
| ADXL345 Accelerometer | 1 mA |
| MPU6050 IMU | 4 mA |
| Crystalfontz 320x230 Graphic Display | 50 mA |
| Crystalfontz 2x16 Character Display | 2 mA |
| TMP102 Temperature Sensor | 1 mA |
| TMET6000 Light Sensor | 1 mA |
| MCP4725 DAC | 1 mA |
| MCP2764 EEPROM | 3 mA |
| **Total** | **63 mA** |

Based on this conservative analysis, total operating current of the selected peripherals is well below the rated maximum of 260mA, allowing for safe operation using the development board's voltage regulator.

Since the LM4F232 development board provides multiple on-chip implementations for each peripheral interface. The peripheral devices were allocated to on-chip bus interfaces and GPIO ports as shown in Figure 16.

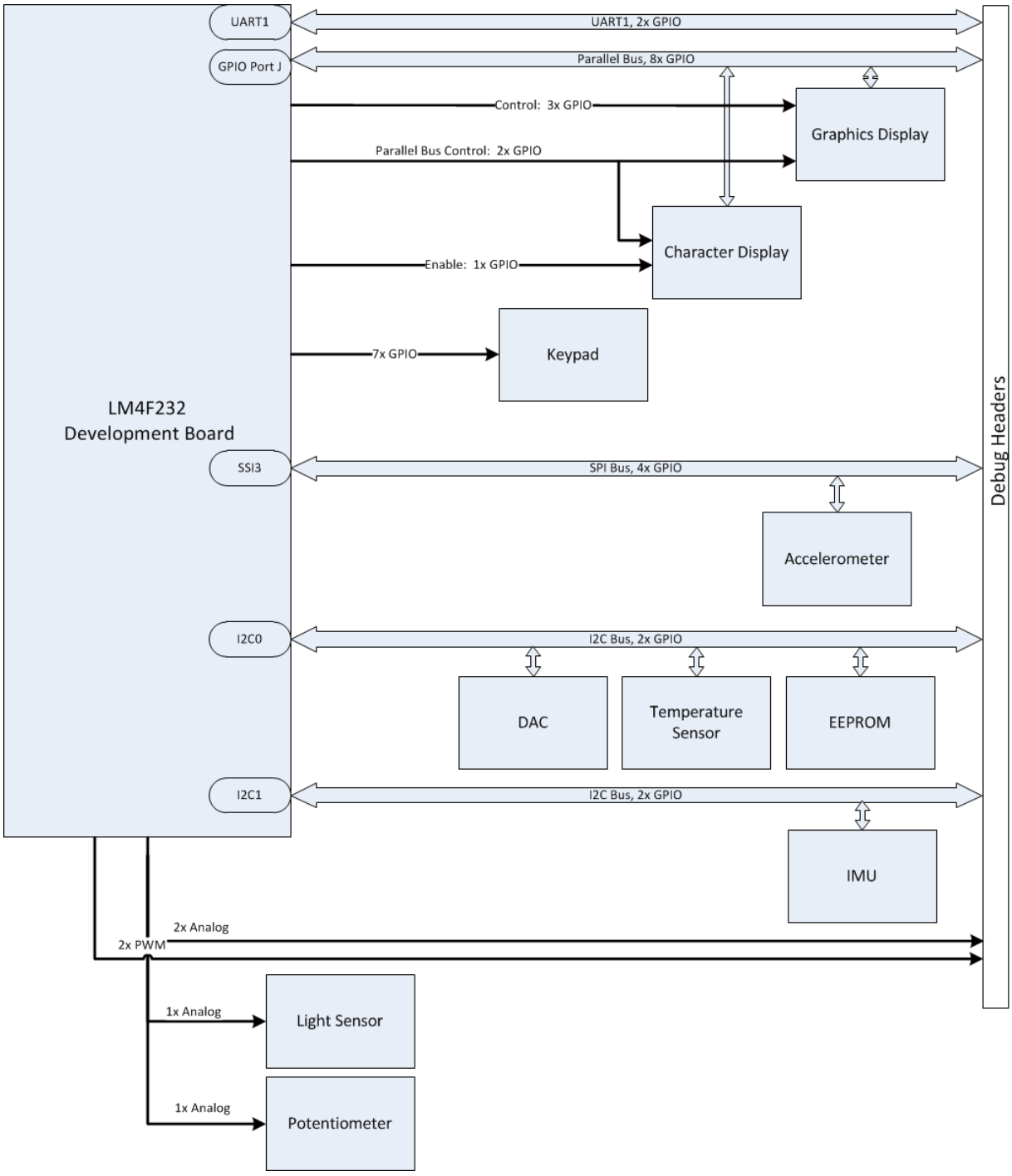



**Figure 16: Peripheral Board Block Diagram with Signal Allocations**

The design considerations for these bus assignments were as follows:

- **Limit congestion on the SPI and I$^2$C buses**: Including too many peripherals on the same bus interface could severely limit performance in real-time labs. Therefore, I allocated only one peripheral that would require near-constant, periodic bus traffic to each available bus.

- **Keep the parallel implementation simple**: As shown in Figure 16, the keypad uses dedicated GPIO signals. While it could share half its signals with the parallel bus, this would have introduced additional parallel bus' circuit complexity and increased the difficulty of a lab involving the keypad. Therefore, the keypad was assigned dedicated GPIO signals to ensure that implementing it would be a feasible student project.

Now that I had selected the microcontroller and peripheral devices for my design, I had realized my system's architecture. My next step was to complete the circuit design and PCB to integrate my design into a single system.

### 4.3.3. PCB Design

Based on the stakeholder analysis from section 4.1, I also identified a set of requirements to dictate the design of my overall system and the PCB. Since students working in lab would be interacting with my PCB as well as the microcontroller and peripheral devices, we decided that my PCB design should:

- **Allow for student interaction with and/or expansion of the hardware interfaces using headers**: ECE 3849 teaches students the fundamental concepts of hardware interfaces and I/O devices. Therefore, my design should make it possible for students to interact with the interfaces provided on my PCB so that they can examine the bus signals on an oscilloscope. This can facilitate students' debugging of their systems and provide the opportunity for students to view and decode data transmissions on the various bus interfaces and observe bus timings, which are important concepts for learning hardware interfaces. To accommodate this requirement, I included headers on my PCB for each bus signal as well as any unused I/O devices available on the microcontroller to support additional labs involving new components.
- **Use verbose labeling on PCB silkscreen to facilitate student usability**: Based on the above requirement, my design provided a multitude of headers to expose the signals on the PCB. In order to encourage students to utilize these signals in their labs, my design should include labeling on the silkscreen for each signal. In this way, students can easily identify signals they need to debug their labs without needing to find this information from another document.

Using these requirements and the components selected in sections 4.3.1 and 4.3.2, I designed a complete schematic to integrate these components and headers onto a single PCB. The circuit design and PCB layout process were conducted using Multisim and Ultiboard 11. As discussed in section 4, my circuit and PCB design process took place in two revisions, termed Revision A and Revision B, respectively, with Revision B being the most recent.

### 4.3.4. Peripheral Board Circuit Design

My full schematics for Revisions A and B are shown in Appendices E and F, respectively. Most of the schematic design was very straightforward due to the usage of pre-built modules and through-hole components. Therefore, many of the peripherals were simply integrated by connecting their bus signals as shown in the block diagram in Figure 16. This was the case for the keypad, displays, and accelerometer, while the $I^2C$ peripherals required determination of pull-up resistors, as discussed below. The notable design considerations for the PCB include the following:

- **Power**: Table 4 in section 4.3.2 demonstrated that the operating current of the peripherals is well below the maximum current provided by the development board on its expansion headers, meaning that my design could utilize the voltage regulator on the LM4F232 development board without additional circuitry.
  In addition, testing confirmed that the LM4F232 development board provides minimal short-circuit protection to shut down the regulator if the power rails are shorted. This means that my design included the same protection—which is very useful for a lab platform used by students—without the need for additional circuitry.
- **I2C Pull-up resistors:** The $I^2C$ bus configurations required external pull-up resistors to keep the bus in an active-high configuration, as dictated by the $I^2C$ protocol specification (Phillips, 2000, p. 14). The specification also dictates that the pull-up resistors should have values between 2kΩ and 20kΩ based on the supply voltage and input current of the devices (p. 15). Since the input current of the $I^2C$ devices varied from a few microamps to 4mA, a conservative value of 10kΩ was selected for each pull-up resistor. The functionality of these values was confirmed during the testing phase, described in section 4.4. The $I^2C$ bus configuration with the pull-up resistors is shown in Figure 17.

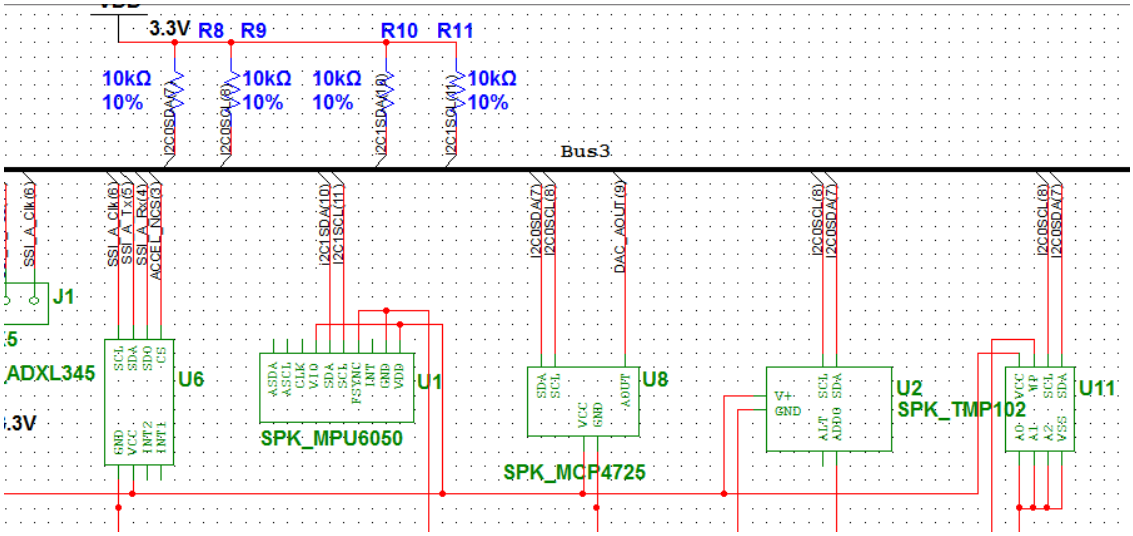



Figure 17: $I^2C$ Bus Configuration

- **Headers:** In accordance with the design requirement to provide headers for viewing the bus signals as well as other unused signals for future expansion, my design included headers for the following on-chip buses and peripherals:
    - Two SPI buses, one including the ADXL345 accelerometer
    - Two $I^2C$ buses, including the DAC, EEPROM, and temperature sensor on one bus and the IMU on the other
    - 8 bit parallel bus for the character and graphics display, including seven control signals to support their respective interfaces
    - 1 UART header to allow for future expansion
    - 3 PWM outputs
    - 1 Analog comparator, which are currently used by ECE 3849's labs as an I/O demonstration
    - 3 Capture/Compare Timers, which are also currently used by ECE 3849's labs
    - 2 Analog inputs to provide additional opportunities for analog measurement
    - 2 GPIO pins to provide digital input/output or chip selects for the SPI bus; all other signals on headers can also be used as GPIO devices
- **CAN Transceiver:** The LM4F232 development board provides on on-chip CAN controller like the current LM3S8962 development board; however, it does not provide an onboard header for connections to other CAN devices. Since one of ECE 3849's labs in the first two course offerings utilized the CAN bus as an I/O device demonstration, I decided to include a CAN header on my design to provide this functionality. This functionality only required an additional header for the CAN bus connector as well as a transceiver IC (available in a DIP package). Notably, the CAN transceiver required an operating voltage of 5V due to the CAN bus' specifications, instead of the 3.3V operating voltage used by the other peripherals. To facilitate this, an additional connection was added to the LM4F232 development board's 5V rail, which was provided on a separate header to the standard expansion ports. The CAN transceiver and header were connected in a similar manner to the Stellaris LM3S8962's CAN configuration, as shown in Figure 18, to allow interaction with the same peripherals. While this complicated the circuit design slightly, it allowed my design to provide an additional peripheral device for use in ECE 3849's labs.

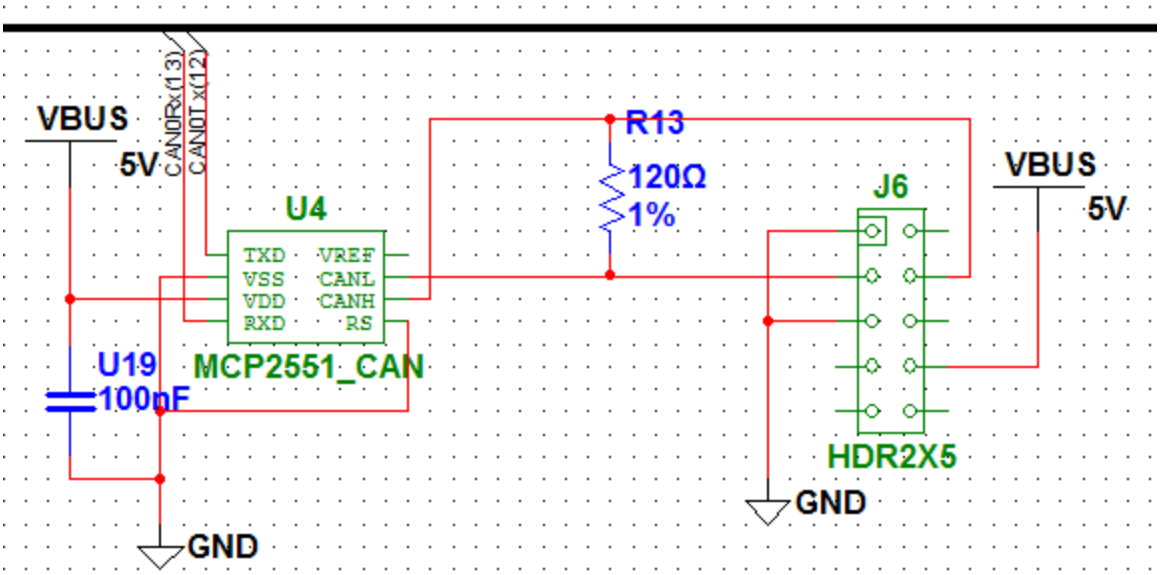



Figure 18: CAN Transceiver Configuration

### 4.3.5. Peripheral Board PCB Layout

In accordance with the usability requirements and schematic defined in the previous sections, I developed a PCB to integrate my design into a single system. My full PCB layouts for revisions A and B are provided in Appendices G and H, respectively. The layout process was conducted in Ultiboard 11. The most notable design challenge for this process was creating custom footprints for the breakout boards in the design, which do not use standard component packages—this required careful measurements to ensure that each component was given enough mechanical clearance on the PCB for installation.

The physical specifications for the PCB were determined based on those that were available for manufacture in a small quantity for prototyping. The PCB manufacturer Advanced Circuits was selected as the manufacturer for the design, as they provide a low-quantity service at a cost of $33 per PCB (Advanced Circuits, 2013). While other manufacturers offer similar promotions, Advanced Circuits was selected based on recommendations from other students and instructors who have used the "$33 each" service in other course projects. To meet their service's specifications, both revisions were created using two copper layers and a single-layer silkscreen. To provide a set of general guidelines for trace routing and PCB layout, I consulted TI's PCB design guidelines for Stellaris microcontrollers, as well as a series of tutorials on using Ultiboard (Guy, 2013; Bitar, 2008). Other notable design considerations for the PCB included the following:

- Power and ground planes to reduce signal noise
- Mounting holes in the corners and center of the board to allow for stability when used by students

- Single-row headers for bus signals to reduce the chance of shorting two rows when connecting oscilloscope probes
- 10mil (0.010”) traces were used for all signals. Since the current usage of each peripheral was very low (as shown in Table 4), none of the traces were required to be wider than a few mil to provide enough power to the peripherals (Advanced Circuits, 2013b). While the minimum trace width specified by the manufacturer was 6mil, a standard width of 10mil was used to provide a margin of error over the manufacturer requirements.

Using these specifications, I created two PCB revisions. The first revision, Rev A shown in Figure 19 was created based on the initial schematic and then tested component-by-component to identify any necessary modifications for a final design.

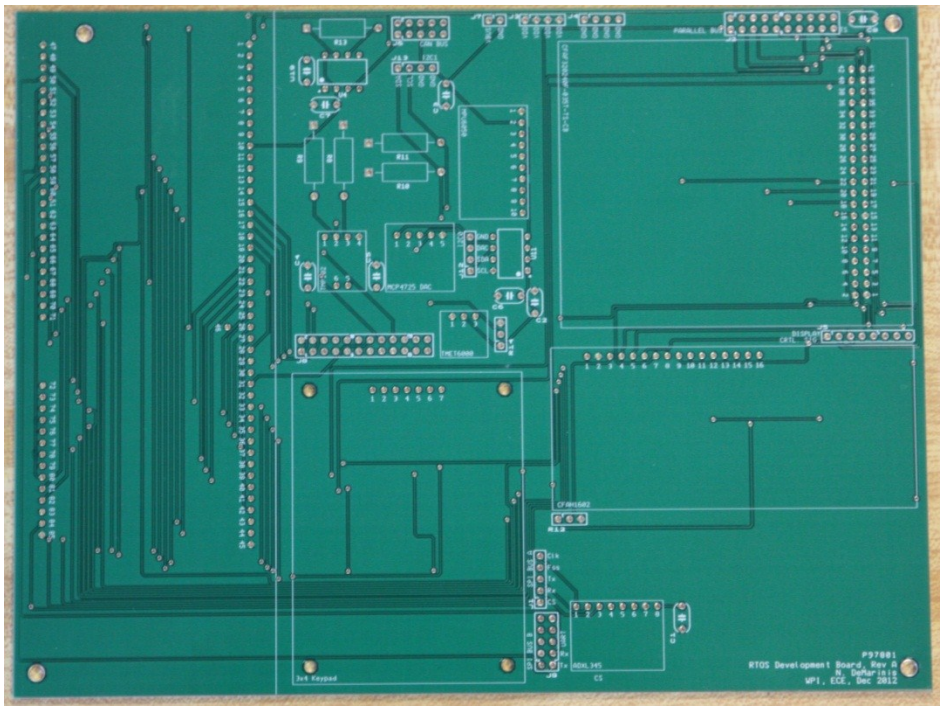


Figure 19: Revision A PCB Layout

## 4.4. System Testing and Verification

Once I had developed a PCB for the system, I need to test the peripherals on my design to verify their functionality and ensure they could be utilized as intended in student labs. This included verification of the electrical specifications of the PCB as well as writing software to control the peripherals as students or instructors would in labs. My testing process included the following phases:

- Mechanical and electrical testing to verify PCB meets schematic design and that all components physically fit on the PCB

- Component-level testing of each peripheral on the development board by verifying electrical functionality and writing isolated demonstration code
- Integrated testing by developing a software project to demonstrate all peripherals on the development board, ensuring functionality of shared buses

Testing for these phases was conducted using a mix of hardware and software debugging provided by standard hardware test equipment (oscilloscope, multimeter) as well as an extensive use of the CCS debugger to identify how the microcontroller was interacting with the peripheral devices. Writing the demonstration code included development of a self-contained demonstration project for each peripheral. To ensure that my initial demonstration code could be used by instructors or students in the future, I encapsulated the code I wrote to interface with each device into common functions that could apply to other labs. While development of an exhaustive software library for this board was well beyond the scope of this project, this provided a simple way to allow extension of my demonstration code by instructors and/or students. In addition, I maintained my code in Git to provide version control and facilitate any future revisions by instructors or students.

### 4.4.1. Physical & Electrical testing results

Physical and electrical testing included two components:

- **Continuity testing:** This entails verifying the electrical connections from the schematic against the PCB to identify any manufacturing defects.
- **Installation Testing:** Once each PCB passed continuity testing, each component was individually installed on the board (by soldering its headers and then connecting the component) and again continuity checked before installing the next component. In the case of a short circuit or faulty solder point caused during installation, incremental testing in this manner helped to isolate the issue to a specific component. In addition, this phase highlighted any issues with the physical device footprints preventing device installation.

My revision A PCB is shown in Figure 20 before and after the installation of the peripheral devices.

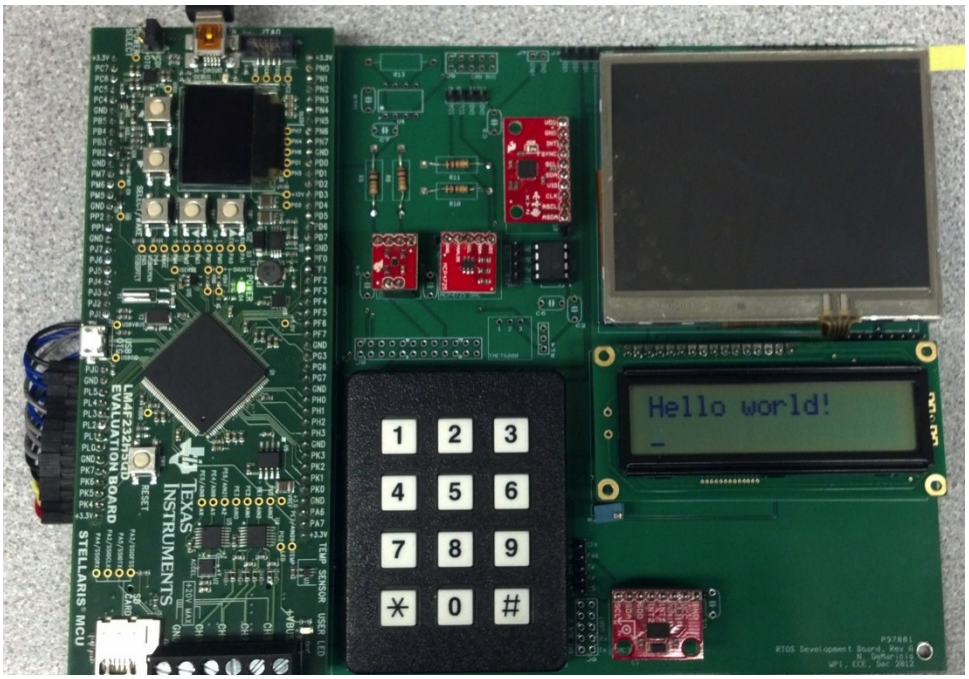



Figure 20: Peripheral Board Revision A with reworks

My physical and electrical testing was mainly successful. Both Revisions A and B were tested in this manner; no defects were found that caused the PCBs tested to deviate from the schematic. When testing Revision A, verification of the device footprints and other mechanical features on the board identified the following issues for correction in Revision B:

- **LM4F232 development board footprint misaligned:** One set of holes for the LM4F232 development board on the PCB was shifted 0.1" from its desired position, meaning that 14 of the 85 pins could not be connected using the sockets. I designed a workaround for this issue using jumper cables (visible in the lower left corner of Figure 20) to allow further prototyping with Revision A. This issue was corrected in Revision B.
- **Additional mounting holes required in center of board and to support peripherals:** Revision A included four mounting holes in each corner of the PCB. However, due to the weight of the components and size of the PCB, additional mounting holes were desired to prevent an implementation deployed in lab from flexing in the center. Three additional mounting holes were added in Revision B.
  Additionally, many of the peripheral devices in Revision A could only be supported by their header connections to the PCB. More mounting holes for standoffs were added in Revision B to allow for additional support of these components.
- **Move headers to edge of board for easier access:** In Revision A, the debug headers for the analog signals and PWM outputs were placed in the center of the board, surrounded by other peripherals, which made the header difficult to access. Since this would de-

crease the PCB's usability by students in lab, these headers were moved to the edges of the board in Revision B.

### 4.4.2. Component-level testing results

My component-level tests are summarized in Table 5, including an explanation of the results and how this affected my design revisions and recommendations for future work.

Table 5: Component-level testing results

| Component under test | Test functionality | Implications |
|---|---|---|
| **MCP4725 Digital-Analog Converter** | • Draws a periodic sawtooth wave at varying frequency | • Confirms functionality of $I^2C$ bus 0<br>• Confirms DAC conversion capability, as would be used in a lab |
| **MCP27AA64 EEPROM** | • Writes a known byte sequence of fixed size to a given address<br>• Reads back previously-written values and verifies against known sequence<br>• Increments known sequence to ensure each subsequent read/write test is valid | • Confirms functionality of $I^2C$ bus 0<br>• Confirms read and write capabilities of EEPROM device |
| **TMP102 Temperature Sensor** | • Reads current temperature and converts to Celsius<br>• Temperature increases/decreases due to environment changes or if sensor is touched by a finger | • Confirms functionality of $I^2C$ bus 0<br>• Confirms functionality of temperature measurements |
| **MPU6050 IMU** | • Calibrates gyroscope readings by taking repeated measurements at startup to define offset values<br>• Periodically reads gyroscope following calibration and converts values for X, Y, and Z axes to deg/sec<br>• Rotation values for axes change according to specification if device is tilted on a particular axis | • Demonstrated incorrect IMU logic voltage connection on Rev A; Fixed on Rev B<br>• Confirms functionality of $I^2C$ bus 1<br>• Confirms communication with MPU 6050 and gyroscope functionality<br>• Appears that full motion-processing is unavailable |
| **ADXL345 Accelerometer** | • Polls X, Y, and Z axes and converts values to g's<br>• Acceleration values change according to specification if device is shaken or inverted on a particular axis | • Confirms SPI bus functionality<br>• Confirms functionality of acceleration measurements |
| **Greyhill Keypad** | • Periodically scans keypad matrix for button presses<br>• Draws button press detection results on onboard display | • Confirms matrix keypad functionality<br>• Confirms functionality of on-chip pull-up resistors used for implementing the keypad |
| **Crystalfontz 2x16 Character Display** | • Displays a given string on startup on first line<br>• Displays keypad button presses on second line | • Confirms functionality of parallel bus<br>• Confirms functionality to display characters and switch lines |
| **Crystalfontz 320x240 LCD Display** | • Initializes display using custom driver for StellarisWare graphics library<br>• Fills display with a given color<br>• Draws keypad input at top of display<br>• Moves a box around the center of the display, changing the background color when it hits the display edge | • Identified current spike in digital LCD backlight control which caused device reset. Corrected in Revision B by changing this to a PWM output.<br>• Confirms functionality of parallel bus and graphic display control signals<br>• Confirms usability of StellarisWare graphics library to interface with display<br>• Frame rate noticeably slow when refreshing onboard display |
| **TMET6000 Light sensor** | • Converts sensor voltage using onboard ADC<br>• Discriminates against preset value to identify if sensor is in light or dark environment | • Confirms functionality of onboard ADC<br>• Confirms that light sensor output varies enough for detection in student labs |
| **CAN Transceiver** | • Adapts StellarisWare example test to send "echo" messages to CAN target device (LM3S8962 board was used for this purpose) | • Confirms functionality of CAN transceiver<br>• Confirms ability to interface with another CAN device |

Overall, my component-level tests demonstrated the following:

- **Confirmed basic functionality on all peripherals:** On the Revision A board, the IMU could not be used on the PCB due to the incorrect logic connection; as an alternative, the IMU was tested on a prototyping board using the headers provided on the PCB. This issue was corrected in Revision B, which could demonstrate the functionality listed above for all peripherals.
  In addition, the Revision A board also utilized a GPIO pin to enable and disable the graphic LCD's backlight. However, I observed that triggering the backlight with a GPIO signal caused the microcontroller to reset when it was powered via USB, though did not cause an issue when powered from a wall outlet (which could provide > 500 mA of input current to the onboard regulator). Therefore, I attributed this reset issue to an instantaneous current spike by the LCD backlight and was able to resolve the issue by connecting a PWM output to the backlight signal, which could turn on the backlight gradually using an input with a low duty cycle. This issue was tested on the Revision A board with an additional prototyping board and corrected with the PWM output on Revision B.
- **IMU provides basic functionality, but no on-chip motion processing:** The MPU-6050 advertises onboard motion processing of accelerometer and gyroscope values (Ivensense, 2011). However, their datasheets do not provide freely-available documentation on how to use these features (Rowberg, 2013). In this configuration, the device only allows access to the raw gyroscope and accelerometer values, which allows motion processing on the LM4F232 development board instead. Therefore, while this does decrease the functionality of the peripheral from its specifications, it still allows for labs involving motion processing and navigation, thereby not affecting the overall capabilities of the system.
- **Graphics display functions as expected, but requires further study to understand full capabilities:** My component tests demonstrated the basic capabilities of drawing shapes, colors, and text, to the graphics display. Most notably, I was able to integrate the display with TI's Stellaris graphics library, which provides functionality for drawing fonts, shapes, colors, menus, and a number of other features that can be useful for labs. When using the StellarisWare graphics library, however, the display refreshed slowly enough to be visible to the human eye when updating a large portion of the display. While this is not a major detriment to the display's functionality, it may impact certain labs that would require high frame rates when updating the entire display. Stellaris development boards that utilize similar displays appear to have similar frame rates, meaning that this might indicate a design issue with the graphics library itself (TI, 2013b). While a full analysis of the graphics library is well beyond the scope of this project, it may be worth conducting such an analysis based on the types of labs instructors wish to implement with this peripheral.

### 4.4.3. Revision B Testing Results

Based on my testing results from Revision A, I created Revision B to correct the specified physical spacing issues as well as apply my electrical reworks for the IMU and graphics display. The justifications for each revision are discussed in the previous sections; the changes from Revisions A to B are summarized below:

- Fixed incorrect logic voltage connection on IMU
- Connected backlight enable signal on graphic LCD to PWM output
- Added more mounting holes in center of the board; increased size of mounting holes to 6/32", as specified by the ECE department
- Added mounting holes for standoffs on the accelerometer, character display, graphics display, IMU, DAC, and keypad to provide additional physical support
- Moved parallel bus and analog expansion headers to PCB edges to allow for easier access
- Added more labeling to debug headers to help students find these signals more easily

My revision B PCB is shown in Figure 21; the additional mounting holes are clearly visible. Revision B was tested in a similar manner to revision A, as described above. Most notably, Revision B corrected the three issues that required wiring reworks on Revision A: the misaligned development header, the backlight enable signal on the graphic display, and the incorrect logic connection on the IMU.

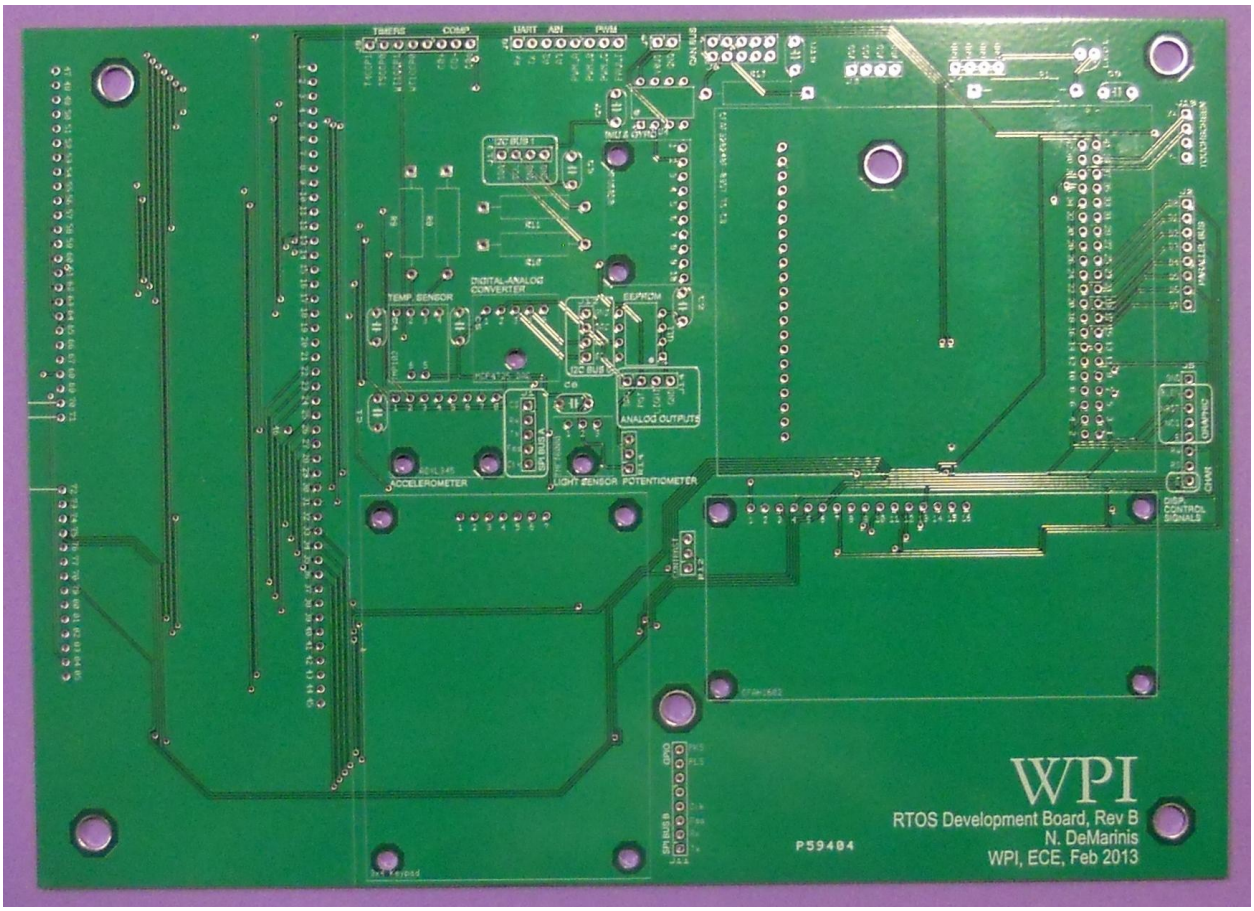



Figure 21: Development Board Revision B PCB

### 4.4.4. Integrated Peripheral Testing Results

Once I had verified the functionality of each individual peripheral, I developed an integrated test to demonstrate the functionality of all peripherals on the board in a single test. This provided a comprehensive demonstration program that could verify that the bus interfaces provided on the development board could handle communication with multiple devices in the same test.

The goal of my integrated test was to simply display the results of all sensors on the development board to the three available displays: the small display on the LM4F232 development board, the character display, and the large graphics display. My integrated test is shown running on the Revision B PCB in Figure 22.

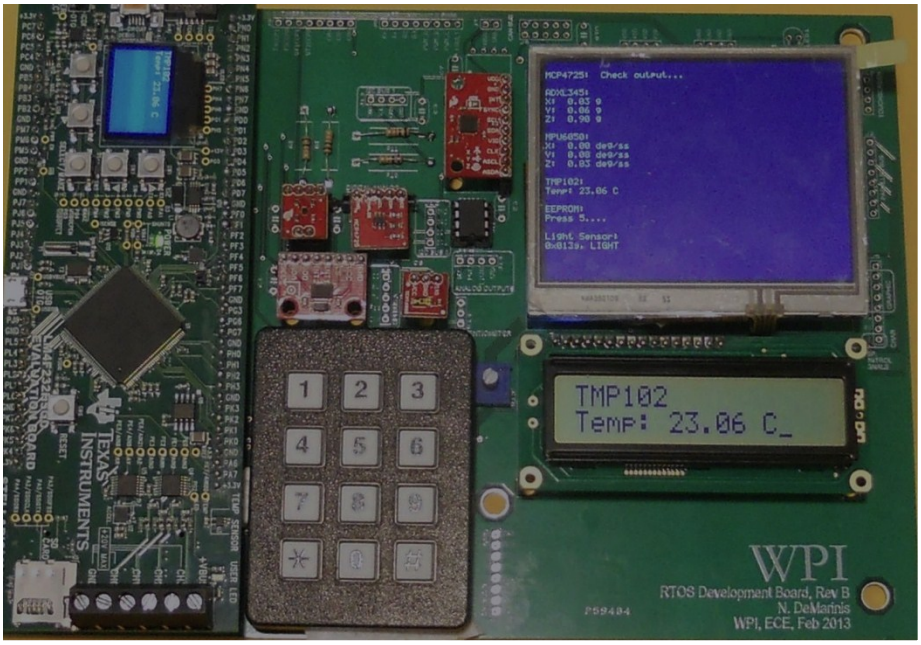



Figure 22: Integrated test program running on Revision B PCB

In this test, the graphics display constantly shows the output of all sensors on the device, which are continually polled for new input. The character display and small graphics display on the LM4F232 board show the output of one sensor, which is selectable by the buttons on the keypad. By polling each sensor and providing the sensor results on all three displays, this test can isolate the failure of a single component in the system. In this way, this integrated test could also be used as a quick diagnostic program to verify the development board's functionality.

## 4.5. Chapter Summary: Development board capabilities

The goal of my development board design was to provide additional hardware capabilities to support new labs that would facilitate student learning in ECE 3849. My design analyzed instructors' and students' requirements for an effective development board for the course and identified and implemented a system based on these requirements.

As requested by instructors, my development board provides variety of peripherals over many peripheral interfaces that demonstrate common tasks in embedded systems design. This can enable labs that provide real-time design challenges for complex software implementations as well as teach students about a variety of I/O devices. My design also speaks to students' usability concerns by providing verbose labeling to help students navigate the hardware.

My specific design goal given by the ECE department was to design a development platform that was feasible to deploy in the computer engineering lab in AK113. As shown by the bill of materials in , the total cost of my board was under the $500 cost target per board. In addition, the board's modular design would allow for my design to resist student abuse over time in labs. Based on this, my design provides a potential solution for a development board that can demon-

strate ECE 3849's course goals. In this way, my project was able to realize a solution targeted to ECE instructors' and students' specific requirements. In order to verify that my design helps facilitate student labs, the development board must be tested by instructors and students. Section 6 outlines a plan for future testing and development to verify this design.

# 5. Investigating challenges to student learning in ECE 3849

This section describes my process for analyzing students' frustrations as they implement their labs in ECE 3849. As discussed in section 2.3, students are often challenged by their lab assignments due to insufficient background in the software and hardware engineering aspects inherent to embedded system design. As a senior tutor for the first two offerings of ECE 3849, I had the opportunity to work with students and identify the specific challenges they face with learning the course material. My goal was to provide targeted materials to help alleviate students' frustrations so they can learn more effectively learn embedded design techniques. To accomplish this, I identified the following objectives:

1. Synthesize student feedback as a tutor for ECE 3849 through informal interviews and focus groups to identify
2. Develop a set of targeted documentation and instructions—a "cookbook"—to that students can use in their labs to help alleviate these challenges

My analysis of student feedback and documentation can provide materials that instructors can give to students taking ECE 3849 to facilitate their lab experiences as well as provide recommendations for instructors to help evolve future course offerings. The following sections describe my process for gathering feedback from students, my analysis of students' challenges, and how these challenges influenced the development of my documentation.

## 5.1. Methods for synthesizing student feedback

In order to identify how students were struggling in ECE 3849's labs, I consulted with students completing their lab assignments as a senior tutor for the course. Senior tutors are undergraduates who typically work with course instructors and teaching assistants to help students with their assignments and lab experiments. As a tutor for ECE 3849, I led weekly help sessions to assist students with their homework and labs, graded homework assignments, and worked with students as they completed their labs during their scheduled lab periods.

During these periods (and often outside of them as well), students would seek out my help with their labs or homework assignments. In addition to asking for help, students would often share their frustrations with their progress on the lab assignments. In these situations, I would often discuss these frustrations with students either in informal, one-on-one interviews or small focus groups. I often asked follow-up questions to get a sense of the specific challenges that impeded their learning of the course material. I focused on asking about students' experiences with the software engineering aspects of the course, particularly the IDE, debugger, and software libraries and documentation, as they represent much of the software complexity added to the ECE curriculum with ECE 3849's introduction.

It is important to note that the feedback I gathered, while useful for pointing out students' challenges, is not an exhaustive survey of students' performance in ECE 3849. The feedback provided by students only includes those students who asked for assistance in the course, which may not be representative of the entire course population. A more comprehensive study of students' course experiences is outlined as a future goal in section 6.

After receiving multitude of feedback and working with students as they completed the course material for two offerings of the course, I identified trends in students' challenges and used them as the focus for my documentation. My analysis of students' challenges is described in section 5.2.

## 5.2. Student challenges to learning in ECE 3849

The feedback I received from students highlighted that students' progress in labs was often impeded by their unfamiliarity with the following aspects of the course:

- **Using software and hardware documentation**: Many students noted that they were unclear about how to go about implementing their labs, despite receiving instructions in their lab assignments. A typical lab assignment for ECE 3849 (an excerpt is shown in Appendix I) provides a high-level overview of the project to complete, describes the hardware and software architecture of the system, and then points students to the documentation so that they can identify how to solve the given problem with the available resources. However, students often expressed difficulty understanding the documentation making comments like, "I've read the lab and the docs, but I don't know what to do," or "I don't know how it can help me." When starting their labs, students are provided a 1400 page hardware datasheet for the microcontroller, over 500 pages of API documentation for the driver libraries, and a number of other documents and example code providing implementation resources (TI, 2010; TI 2012b; TI, 2012c). There is no current "roadmap" for this documentation and it is written for an audience of embedded designers, rather than inexperienced students. It follows that students are overwhelmed the documentation and thus misunderstand how to use it in their labs.
- **Using software libraries**: Most of the code students write for their labs uses a software library provided by TI for interacting with the on-chip peripherals. In many cases, the lab assignments provided "starter code" that indicated which library functions students could use for their labs. Students cited difficulty understanding how to use these library functions in their code, even when provided a function prototype and functional description. This demonstrates students' unfamiliarity with the software engineering challenge of using APIs to accomplish tasks. As discussed in section 2.2, students do not encounter third-party libraries in their background courses, meaning that using them in ECE 3849 is a new experience.

- **Using the IDE and debugger:** Even though the lab assignments provide a basic outline of how to use the advanced IDE features of Code Composer Studio, students noted that they found using the Code Composer Studio IDE very frustrating in their labs. Most notably, the labs require that students use not only the software libraries mentioned above, but also an additional set of drivers and code provided for the lab. This means that students need to understand how to navigate a pre-existing codebase. As students complete more labs, they are required to manage multiple code projects in the IDE, adding another level of complexity. The large codebase and the multitasked software architecture of students' labs also requires that students learn the advanced features of the included debugger to help identify problems with their lab implementations was also cited as a source of students' frustrations.
- **Understanding their build environment, including compiler and linker errors:** As described above, ECE 3849's labs use a complex build environment. Students often complained that they had difficulty configuring their environment and understanding when it was causing errors in their projects. The documentation explaining how students should do this, included as part of students' first lab assignment, spans over pages of dense instructions, which students often found hard to follow. I often helped many students debug compiler and linker errors caused by invalid build configurations: it was evident that students did not fully understand how the environment was configured, leading to their confusing when presented with an error related to their environment configuration.

While all of these challenges are important software engineering concepts, they are incidental to the domain-specific embedded systems concepts central to ECE 3849. Students need to understand how to perform tasks like configuring the build environment, using the IDE, and using the library documentation, in order to complete their labs or develop software for an embedded system in a professional environment. Therefore, the focus of my documentation was to create a "cookbook" to teach students how to perform these critical tasks to help them focus on the embedded systems concepts they need to demonstrate in their labs.

## 5.3. Designing a "cookbook" for ECE 3849

As discussed in section 5.2, my goal for supporting documentation was to provide a set of instructions for accomplishing the critical but incidental software engineering tasks inherent to ECE 3849's labs. I provided this in the form of a "cookbook" outlining how students can accomplish these tasks, which is included in Appendix J.

To identify the topics my cookbook should cover, I examined the challenges I outlined in section 5.2 and mapped these to a set of use cases and other key points that could demonstrate how students could navigate the development tools and documentation.

These use cases and key points, shown in Table 6, became the basis for the topics covered in my cookbook.

**Table 6: Cookbook topics to alleviate students' course challenges**

| Identified Student Challenges | Related use cases and key points covered in cookbook |
|---|---|
| **Using software and hardware documentation** | • Overview of required software and documentation for development<br>• Useful documentation and examples<br>• Navigating the documentation |
| **Using software libraries** | • Using the StellarisWare documentation<br>• Using function documentations |
| **Using the IDE** | • Navigating CCS's UI perspectives<br>• Creating and using projects and workspaces<br>• Importing example code projects |
| **Using the debugger** | • Building and executing code in the debugger<br>• Setting breakpoints<br>• Viewing variables |
| **Configuring the build environment** | • Creating your own projects |
| **Understanding hardware peripherals** | • Useful documentation and examples<br>• Using GPIO peripherals |
| **Understanding compilation errors** | • Troubleshooting common errors |

Using these topics as an outline, I created the cookbook based on the design considerations listed above. Since a major finding of my analysis was that students are frequently overwhelmed by the available documentation, any document that I created to help needed to be highly accessible to students. Based on this, I decided to take the following design approach when explaining the topics in my "cookbook" to keep students engaged, reduce ambiguity, and provide clear connections to their lab assignments. My design considerations for the cookbook were as follows:

- Use frequent illustrations and examples to demonstrate topics
- Demonstrate concepts using step-by-step procedures
- Organize the document by common use cases, rather than features, to allow students to find information easily

Most notably, the document heavily utilized screenshots from the IDE with annotations to demonstrate concepts, such as the excerpt shown in Figure 23.

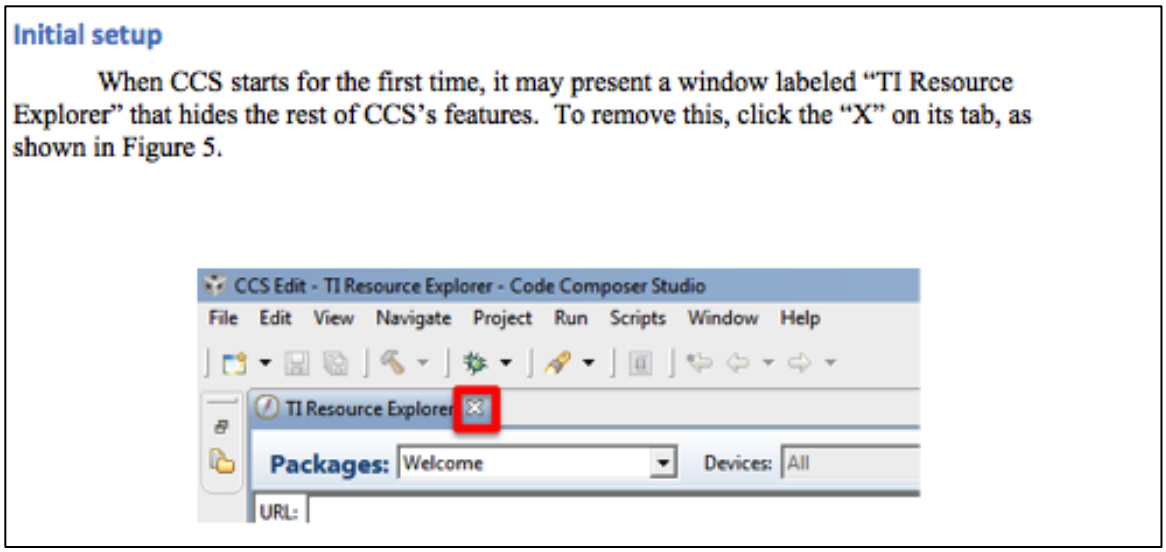



Figure 23: Cookbook excerpt with annotated screenshot

My final cookbook document (included in Appendix J) encompassed 44 pages of documentation, including the following major sections:

- **Using Code Composer Studio**: This section provides a comprehensive introduction to the main user interface of code composer and provides step-by-step instructions for configuring a build environment for use with my development board and the existing course software libraries. It also includes instructions for common tasks using the debugger, like watching variables and setting breakpoints in multiple files.
- **Navigating the documentation**: This section includes a high-level overview of the two major documents that provide information for students' lab implementations: the microcontroller datasheet, and the StellarisWare software library. It provides an outline of the most useful sections of these documents for providing high-level overviews of the hardware and software functionality. The description of the software documentation also explains how students can commonly use function prototypes to understand library functions.
- **Using GPIO Peripherals:** This section provides a high-level overview of how the Stellaris microcontroller uses GPIO peripherals, which was confusing for many students at the beginning of the course. While this section only describes GPIO peripherals, it can serve as a template for future sections that describe more hardware components in a similar manner.
- **Useful documentation and examples:** This section provides a brief overview of the multitude of documents and example code available to help students implement their labs. It describes what each document contains and how it could be useful to students

- **Troubleshooting:** This section lists some common errors that I encountered when helping students and recommend steps to resolve them. Most importantly, it explains each listed error to help teach students how to interpret such errors.

## 5.4. Chapter summary: Future cookbook iterations

My analysis of student feedback has shown that students are often frustrated by the incidental software and hardware engineering aspects of embedded system design, including IDEs, debugging, and understanding documentation. To help alleviate these challenges, my cookbook documentation provides a set of instructions and examples to help students navigate the software and hardware tools necessary for learning embedded systems design in the course. This provides a potential solution for making the complex embedded systems design concepts more accessible to students taking embedded systems courses.

In order to verify that my design can help alleviate students' challenges in the course, my cookbook must be tested with students and instructors in ECE 3849 to identify how it affects students understanding of the hardware and software development tools. In addition, my cookbook could also provide a starting point for instructors or course staff to extend with additional use cases or information based on further research into students' challenges or common errors identified in new labs. In this way, my cookbook design creates a potential solution for allowing instructors to help reduce students' challenges as the course evolves.

# 6. Discussion and Future Work

This section reviews my project's goals and outlines plans for future development to facilitate student learning of embedded systems design at WPI. This includes a summary of my major deliverables and an outline for future work to utilize my development board and supporting documentation to support students and instructors in ECE 3849.

## 6.1. Summary of deliverables

My overall goal was to develop a comprehensive teaching platform for supporting student labs in ECE 3849. In order to do this, I developed:

- A peripheral board design based on ECE instructors' and students' requirements, including a functioning prototype
- A set of extensible demonstration code to exercise the devices on my peripheral board and provide a starting point for lab implementations
- An analysis of students' major challenges when implementing their laboratory assignments in ECE 3849 to serve as recommendations for future course offerings
- A set of targeted documentation, or "cookbook," to help alleviate these specific challenges

In this way, my development platform contains hardware, software, and pedagogical components to provide a potential solution for improving student learning in ECE 3849 and other embedded systems courses at WPI. Each of these components provide recommendations for helping students learn embedded systems course concepts, either by providing hardware to facilitate new and engaging labs, or with documentation to help students complete their labs more effectively. I have presented my design to the ECE department for review by the curriculum committee, who will decide how my work may be integrated into the existing curriculum.

## 6.2. Next Steps: an outline for future work

Should the ECE department decide to move forward with integrating my development board or documentation into the embedded systems curriculum, I have provided the following recommendations for ensuring that platform provides an effective solution:

- **Investigate robust mounting solutions for the peripherals:** At present, many of the peripheral devices are attached to the board using only their headers. While I have provided mounting holes in my revision B PCB, further research is needed to identify a usable set of headers, standoffs, and sockets to secure the peripherals to the development board. This is necessary to ensure that my design can be feasibly deployed in AK 113.

- **Develop additional software tests to exercise the peripherals in an example lab configuration:** My demonstration code was designed to verify the functionality of the peripheral devices. Additional software tests could demonstrate example labs that could be performed by students using my design to help identify feasible lab assignments for the course.
- **Continue testing the graphics display:** My demonstration code showed that the frame rate on the graphics display was noticeably low when using the StellarisWare graphics library. Further tests could identify the maximum performance capabilities of the display using my design, perhaps using a different software implementation.
- **Investigate integration with development board for ECE 2049:** An additional design was proposed during this project's timeline that envisioned an MSP430-based peripheral board that could use elements of my design to provide ECE 2049's labs. Thus, both platforms could be combined to support labs for both ECE 2049 and ECE 3849. Since ECE 2049's development boards are reaching end-of-life, such an integrated design could provide a very useful solution for both courses.
- **Pilot test my development platform with ECE students taking ECE 3849:** In order to verify that my design facilitates student learning in ECE 3849, my design (using the development board, the cookbook, or both) could be tested during an offering of ECE 3849 with a limited number of students. By surveying students before and after the lab or holding discussions, the ECE department could obtain student feedback on my design's usability.
- **Extend the cookbook with common issues students encounter in their labs:** One of the components of my cookbook is a "Troubleshooting" section that explains some common compiler and linker errors and provides suggestions for resolving them. In future offerings of ECE 3849 using my cookbook, instructors and course staff could extend this section with any common student challenges students find in the course. These extensions could include descriptions of the problem, recommended solutions, and, most importantly, an explanation of the error or task to help students understand the issue.

## 6.3. Final discussion

The goal of my project was to develop a comprehensive teaching platform for students learning embedded systems design in ECE 3849. In order to identify how to best facilitate student learning in the course, I identified the pedagogical challenges in learning embedded systems design concepts by analyzing the course objectives and by working with students as a tutor in their labs. My design includes a hardware platform based on instructors' and students' requirements to provide additional hardware capabilities and enable engaging labs that help students learn the diverse range of topics in ECE 3849. My "cookbook" documentation provides a potential solution to students' frustrations with the software and hardware engineering aspects inherent to em-

bedded systems. Thus, my overall design integrates both hardware and software solutions to help students learn the how to design embedded systems as part of WPI's computer engineering curriculum.

# 7. Works Cited


Advanced Circuits. (2013). *2-Layer PCB Designs*. Retrieved 18 April, 2013, from http://www.4pcb.com/33-each-pcbs/index.html

Advanced Circuits. (2013). *PCB Trace Online Calculator*. Retrieved Apr 18, 2013, from http://www.4pcb.com/trace-width-calculator.html

Bitar, S. J. (2008). *PCB Layout Details*. Retrieved Nov 2012, from http://ece.wpi.edu/~sjbitar/misc/

Christof Ebert, C. J. (2009). Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42-52.

Davies, J. (2008). *MSP430 Microcontroller Basics.* Burlington: Elsevier.

Future Technology Devices International. (2012). *UM232H FT232H Development Module Datasheet*. Retrieved Apr 18, 2013, from http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_UM232H.pdf

Guy, J. (2013). *System Design Guidelines for Stellaris Microcontrollers*. (Texas Instruments) Retrieved from http://www.ti.com/lit/an/spma036b/spma036b.pdf

Ivensense Inc. (2011). *MPU-6000 and MPU-6050 Product Specification*. Retrieved Dec 2012

Jet propulsion Laboratory. (2013). *Mission*. Retrieved Apr 18, 2013, from Mars Science Laboratory: Curiosity Rover: http://mars.jpl.nasa.gov/msl/mission/rover/

Kamal, R. (2006). *Embedded Systems: Architecture, Programming, and Design.* McGraw-Hill.

Klaus Havelund, A. G. (2009). Monitoring the Execution of Space Craft Flight Software. *The European Joint Converences on Theory and Practice of Software.* York, UK.

Mikroelekrtonika. (2013). *Mikromedia for Stellaris M3*. Retrieved Apr 18, 2013, from http://www.mikroe.com/mikromedia/stellaris-m3/

Miller, J. (2012, Aug 6). *Wind River's VxWorks powers Mars Science Laboratory Rover, Curiosity*. Retrieved Apr 18, 2013, from Wind River Newsroom: http://www.windriver.com/news/press/pr.html?ID=10901

Motavalli, J. (2010, Feb 4). *The Dozens of Computers that make Modern Cars Go (and Stop).* Retrieved Apr 18, 2013, from http://www.nytimes.com/2010/02/05/technology/05electronics.html

Phillips Semiconductors. (2000). *The I2C-bus specification*. Retrieved Apr 18, 2013, from http://www.classic.nxp.com/acrobat_download2/literature/9398/39340011.pdf

Rowberg, J. (2013). *MPU-6050 6-axis accelerometer/gyroscope*. Retrieved Apr 18, 2013, from I2CDevlib: http://www.i2cdevlib.com/devices/mpu6050

Sparkfun. (2013). *Triple Axis Accelerometer Breakout - ADXL345*. Retrieved Apr 18, 2013, from https://www.sparkfun.com/products/9836

Texas Instruments. (2010). *LM3S9B96 Datasheet.* Retrieved Apr 18, 2013, from http://www.ti.com/product/lm3s9b96

Texas Instruments. (2010). *Stellaris LM3S8962 Evaluation Board.* Retrieved 9 04, 2012, from http://www.ti.com/tool/ek-lm3s8962

Texas Instruments. (2012). *EKS-LM4F232 Evaluation Kit with Code Composer Studio*. Retrieved Apr 18, 2013, from http://www.ti.com/tool/eks-lm4f232

Texas Instruments. (2012). *Stellaris Launchpad Evaluation Kit*. Retrieved Oct 2012, from http://www.ti.com/tool/ek-tm4c123gxl

Texas Instruments. (2012). *Stellaris LM4F232H5QD Microcontroller Datasheet*. Retrieved Nov 2012

Texas Instruments. (2012). *Stellaris Peripheral Driver Library*. Retrieved Dec 2012

Texas Instruments. (2013). *Code Composer Studio Integrated Development Environment (IDE) v5*. Retrieved Apr 18, 2013, from http://www.ti.com/tool/ccstudio

Texas Instruments. (2013). *LM3S9B96 Development Kit*. Retrieved Apr 18, 2013, from http://www.ti.com/tool/dk-lm3s9b96

Wind River. (2011). *Wind River VxWorks Platforms 6.9 Overview.* Retrieved Apr 10, 2012, from http://www.windriver.com/products/product-overviews/PO_VE_6_9_Platform_0211.pdf

Worcester Polytechnic Institute. (2012, 1 12). *Undergraduate Courses*. Retrieved 10 1, 2012, from Electrical Computer Engineering Department: http://www.wpi.edu/academics/ece/ugcourses.html

Worcester Polytechnic Institute. (2012). *Undergraduate Courses*. Retrieved Apr 18, 2013, from Computer Science Department: http://www.wpi.edu/academics/cs/ugcourses.html

WPI ECE Department. (n.d.). *MSP430 Expansion Board Design*. Retrieved 10 1, 2012, from http://ece.wpi.edu/msp430

# Appendix A. Digital ECE Course Descriptions

## ECE 2029. INTRODUCTION TO DIGITAL CIRCUIT DESIGN

Cat. I Digital circuits are the foundation upon which the computers, cell phones, and calculators we use every day are built. This course explores these foundations by using modern digital design techniques to design, implement and test digital circuits ranging in complexity from basic logic gates to state machines that perform useful functions like calculations, counting, timing, and a host of other applications. Students will learn modern design techniques, using a hardware description language (HDL) such as Verilog to design, simulate and implement logic systems consisting of basic gates, adders, multiplexers, latches, and counters. The function and operation of programmable logic devices, such as field programmable gate arrays (FPGAs), will be described and discussed in terms of how an HDL logic design is mapped and implemented. Experiments involving the design of combinational and sequential circuits will provide students a hands-on introduction to basic digital electrical engineering concepts and the skills needed to gain more advanced skills. In the laboratory, students will construct, troubleshoot, and test the digital circuits that they have developed using a hardware description language. These custom logic designs will be implemented using FPGAs and validated using test equipment. Topics: Number representations, Boolean algebra, design and simplification of combinational circuits, arithmetic circuits, analysis and design of sequential circuits, and synchronous state machines. Lab exercises: Design, analysis and construction of combinational and sequential circuits; use of hardware description languages to implement, test, and verify digital circuits; function and operation of FPGAs. Recommended background: Introductory Electrical and Computer Engineering concepts covered in a course such as ECE 2010 or RBE 2001, and MA 1022. Note: Students will not be able to receive credit for both ECE 2022 and ECE 2029.

## ECE 2049. EMBEDDED COMPUTING IN ENGINEERING DESIGN

Cat. I Embedded computers are literally everywhere in modern life. On any given day we interact with and depend on dozens of small computers to make coffee, run cell phones, take pictures, play music play, control elevators, manage the emissions and antilock brakes in our automobile, control a home security system, and so on. Using popular everyday devices as case studies, students in this course are introduced to the unique computing and design challenges posed by embedded systems. Students will then solve real-world design problems using small, resource constrained (time/memory/power) computing platforms. The hardware and software structure of modern embedded devices and basic interactions between embedded computers and the physical world will also be covered in lecture and as part of laboratory experiments. In the laboratory, emphasis is placed on interfacing embedded processors with common sensors and devices (e.g. temperature sensors, keypads, LCD display, SPI ports, pulse width modulated motor controller outputs) while developing the skills needed to use embedded processors in systems design. This course is also appropriate for RBE and other engineering and CS students interested in learning about embedded system theory and design. Topics: Number/data representations, embedded system design using C, microprocessor and microcontroller architecture, program development and debugging tools for a small target processor, hardware/software dependencies, use of memory mapped peripherals, design of event driven software, time and resource management, applications case studies. Lab Exercises: Students will solve commonly encountered embedded processing problems to implement useful systems. Starting with a requirements list students will use the knowledge gained during the lectures to implement solutions to problems which explore topics such as user interfaces and interfacing with the physical world, logic flow, and timing and time constrained programming. Exercises will be performed on microcontroller and/or microprocessor based embedded systems using cross platform development tools appropriate to the target platform. Recommended Background: ECE 2010 or equivalent knowledge in basic circuits, devices and analysis; and C language programming (CS 2301 or equivalent) Suggested Background: ECE 2029 or equivalent knowledge of digital logic, logic signals and logic operations; Note: Students who have received credit for ECE 2801 may not received credit for ECE 2049.

## ECE 3803. MICROPROCESSOR SYSTEM DESIGN

Cat. I This course builds on the computer system material presented in ECE 2801. It covers the architecture, organization and instruction set of microprocessors. The interface to memory (RAM and EPROM) and I/O peripherals is described with reference to bus cycles, bus timing, and address decoding. Emphasis is placed on the design, programming and implementation of interfaces to microprocessor systems using a mixture of C and assembly language. Topics: bus timing analysis, memory devices and systems, IO and control signaling, bi-directional bus interfaces, instruction execution cycles, interrupts and polling, addressing, programmable peripheral devices, interface design issues including analog/digital and digital/analog conversion. Mixed language (C and Assembler) programming. Laboratory exercises: Use of standard buses for advanced IO design and programming, mixed language programming, standard bus timing, and interface design and implementation. Development of a complete standalone embedded computer system. Recommended background: ECE 2029 and ECE 2049 or an equivalent background in advanced logic design, and microprocessor architecture. CS 2301 or CS 2303 or an equivalent background in C programming.

## ECE 3810. ADVANCED DIGITAL SYSTEM DESIGN

This is an introductory course addressing the systematic design of advanced digital logic systems. The emphasis is on top-down design starting with high level models using VHDL as a tool for the design, synthesis, modeling, and testing of highly integrated digital devices. The integration of tools and design methodologies will be addressed through a discussion of system on a chip (SOC) integration, methodologies, design for performance, and design for test/testing. Topics: 1) hardware description languages, system modeling, synthesis, simulation and testing of digital circuits; 2) design integration to achieve specific SOC goals including architecture, planning and integration, and testing; 3) use of soft core and IP modules to meet specific architecture and design goals. Laboratory exercises: VHDL models of combinational and sequential circuits, synthesizing these models to programmable logic devices, simulating the design, test-benches, system design and modeling, integration of IP and high level SOC design methodologies. Recommended background: ECE 2029 (or ECE 3801), and experience with programming in a high-level language such as C. Suggested background: ECE 3803.

## ECE 3829. ADV DIGITAL SYSTM DESGN W FPGA

This course covers the systematic design of advanced digital systems using FPGAs. The emphasis is on top?down design starting with high level models using a hardware description language (such as VHDL or Verilog) as a tool for the design, synthesis, modeling, test bench development, and testing and verification of complete digital systems. These types of systems include the use of embedded soft core processors as well as lower level modules created from custom logic or imported IP blocks. Interfaces will be developed to access devices external to the FPGA such as memory or peripheral communication devices. The integration of tools and design methodologies will be addressed through a discussion of system on a chip (SOC) integration, methodologies,design for performance, and design for test. Topics: 1. hardware description languages, system modeling, synthesis, simulation and testing of digital circuits; 2. design integration to achieve specific system design goals including architecture, planning and integration, and testing; 3 use of soft core and IP modules to meet specific architecture and design goals. Laboratory exercises: Students will design and implement a complete sophisticated embedded digital system on an FPGA. HDL design of digital systems including lower level components and integration of higher level IP cores, simulating the design with test benches, and synthesizing and implementing these designs with FPGA development boards including interfacing to external devices. Recommended background: ECE2029 and ECE 2049 Notes: Students may not receive credit for ECE3829 if they have received credit for ECE3810.

## ECE 3849. REAL-TIME EMBEDDED SYSTEMS

This course continues the embedded systems sequence by expanding on the topics of realtime software and embedded microprocessor system architecture. The software portion of this course focuses on solving real?world problems that require an embedded system to meet strict real?time constraints with limited resources. On the hardware side,

this course reviews and expands upon all the major components of an embedded microprocessor system, including the CPU, buses, memory devices and peripheral interfaces. New IO standards and devices are introduced and emphasized as needed to meet system design, IO and performance goals in both the lecture and laboratory portion of the course. Topics: Cross?compiled software development, embedded system debugging, multitasking, real?time scheduling, inter?task communication, software design for deterministic execution time, software performance analysis and optimization, device drivers, CPU architecture and organization, bus interface, memory management unit, memory devices, memory controllers, peripheral interfaces, interrupts and interrupt controllers, direct memory access. Laboratory exercises: Programming real?time applications on an embedded platform running a real?time operating system (RTOS), configuring hardware interfaces to memory and peripherals, bus timing analysis, device drivers. Recommended background: ECE 2029 and ECE 2049.

## ECE 4801. ADVANCED COMPUTER SYSTEM DESIGN

Cat. I This course continues the development of advanced computer systems and focuses on the architectural design of standalone embedded and highperformance microprocessor systems. Topics: advanced microprocessor architecture, embedded systems, RISC and CISC, interrupts, pipelining, DMA, cache and memory system design, highperformance system issues. Recommended background: ECE 3803, ECE 4301 or equivalent.

# Appendix B. Computer Science Background Course Descriptions

## CS 1101. INTRO TO PROGRAM DESIGN

Cat. I This course introduces principles of computation and programming with an emphasis on program design. Topics include design and implementation of programs that use a variety of data structures (such as records, lists, and trees), functions, conditionals, and recursion. Students will be expected to design, implement, and debug programs in a functional programming language. Recommended background: none. Either CS 1101 or CS 1102 provide sufficient background for further courses in the CS department. Undergraduate credit may not be earned for both this course and CS 1102. Undergraduate credit may not be earned both for this course and for CS 2135.

## CS 1102. ACCELERTD INTRO TO PROGR DESGN

Cat. I This course provides an accelerated introduction to design and implementation of functional programs. The course presents the material from CS 1101 at a fast pace (so students can migrate their programming experience to functional languages), then covers several advanced topics in functional programming (potential topics include macros, lazy programming with streams, and programming with higher-order functions). Students will be expected to design, implement, and debug programs in a functional programming language. Recommended background: prior programming background covering lists, trees, functions, and recursion. Undergraduate credit may not be earned for both this course and CS 1101. Undergraduate credit may not be earned both for this course and for CS 2135.

## CS 2301. SYS PROGR FOR NON-MAJORS

Cat. I This course introduces the C programming language and system programming concepts to non-CS majors who need to program computers in their own fields. The course assumes that students have had previous programming experience. It quickly introduces the major concepts of the C language and covers manual memory management, pointers and basic data structures, the machine stack, and input/output mechanisms. Students will be expected to design, implement, and debug programs in C. Recommended background: CS 1101 or CS 1102 or previous experience programming a computer. All Computer Science students and other students wishing to prepare for upper-level courses in Computer Science should take CS 2303 instead of CS 2301. Students who have credit for CS 2303 may not receive subsequent credit for CS 2301.

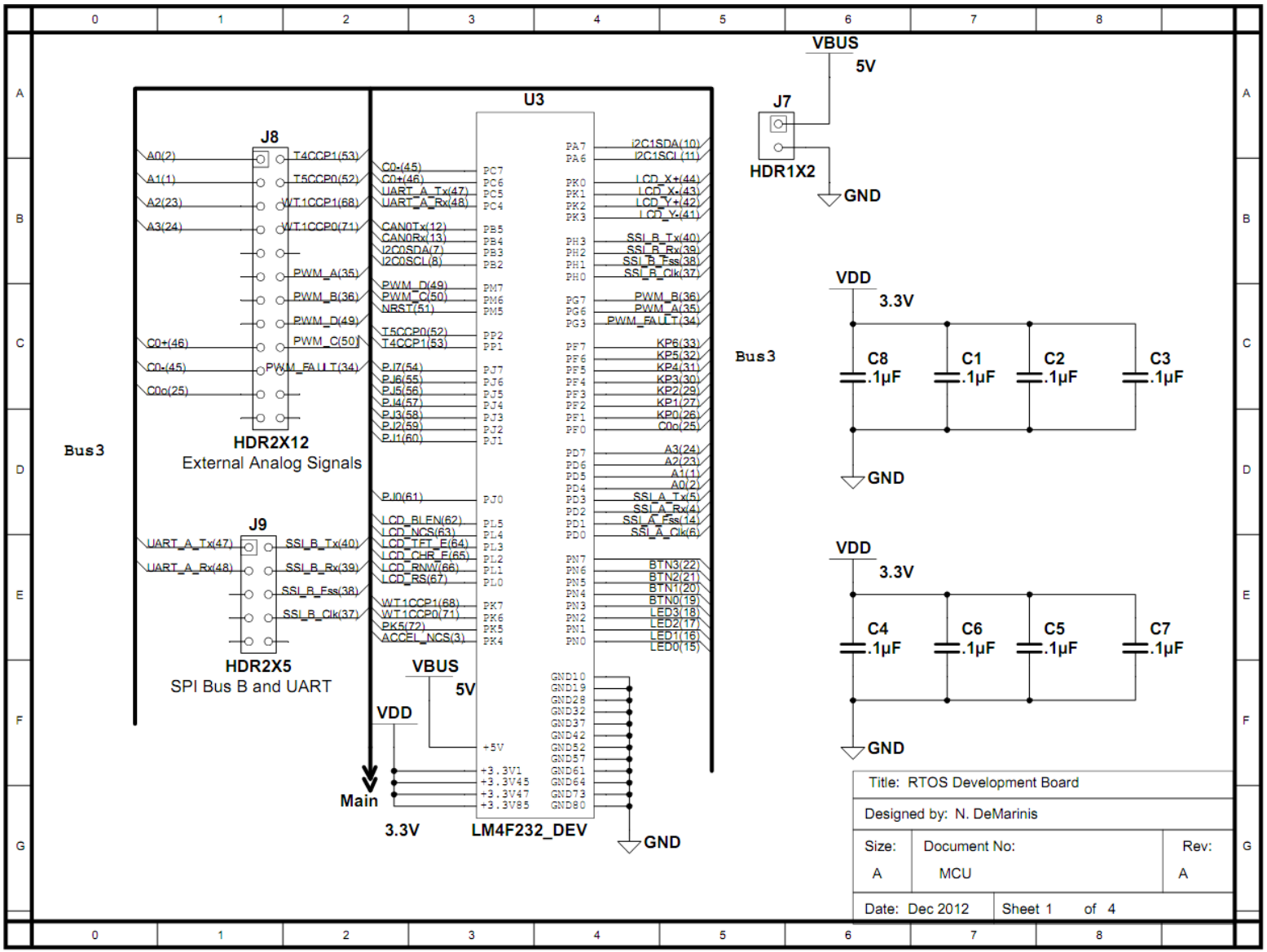
## CS 2303. SYSTEMS PROGRAMMING CONCEPTS

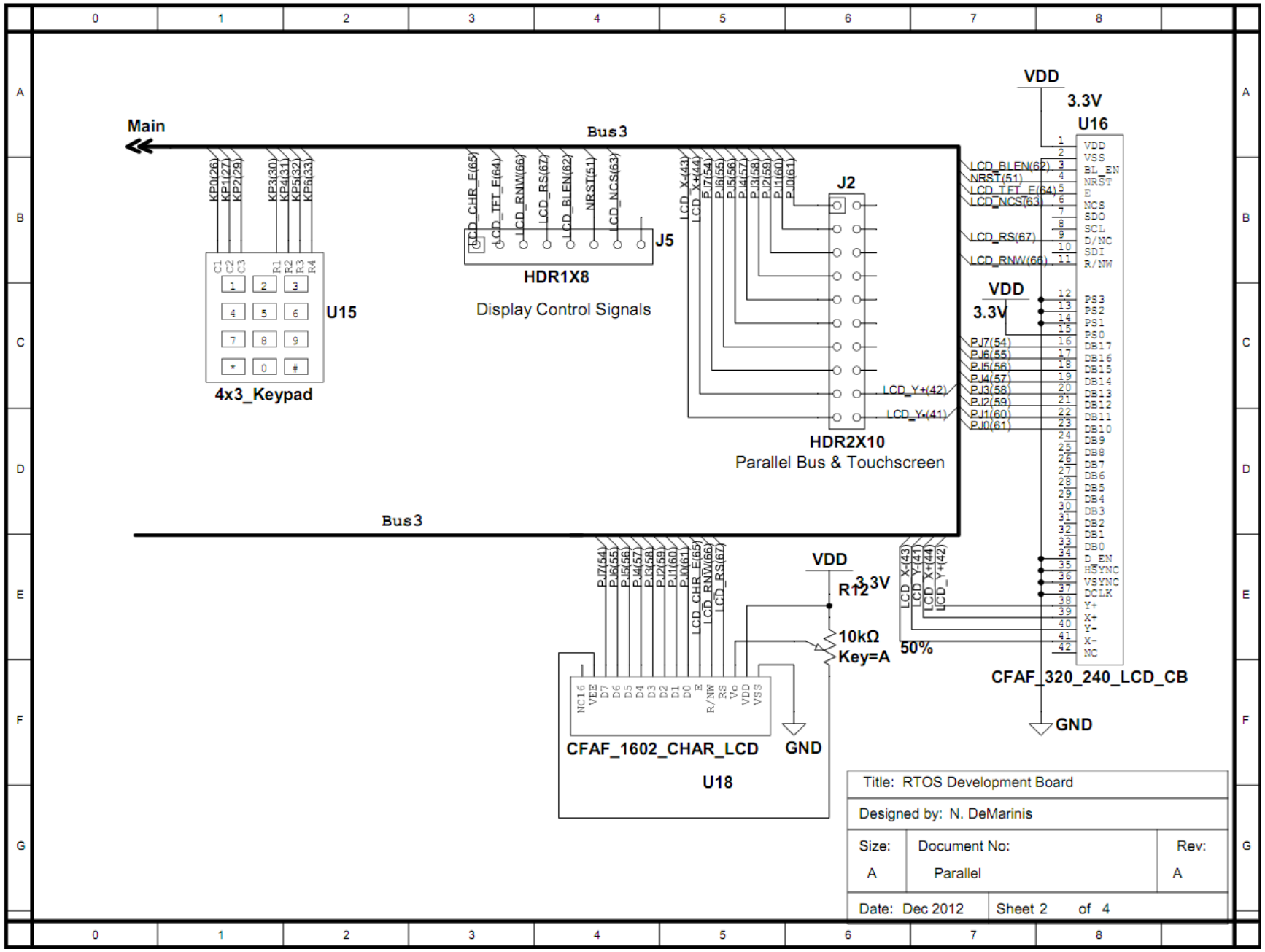
Cat. I This course introduces students to a model of programming where the programming language exposes details of how the hardware stores and executes software. Building from the design concepts covered in CS 2102, this course covers manual memory management, pointers, the machine stack, and input/ output mechanisms. The course will involve large-scale programming exercises and will be designed to help students confront issues of safe programming with system-level constructs. The course will cover several tools that assist programmers in these tasks. Students will be expected to design, implement, and debug programs in C++ and C. The course presents the material from CS 2301 at a fast pace and also includes C++ and other advanced topics. Recommended background: CS 2102 and/or substantial object-oriented programming experience.

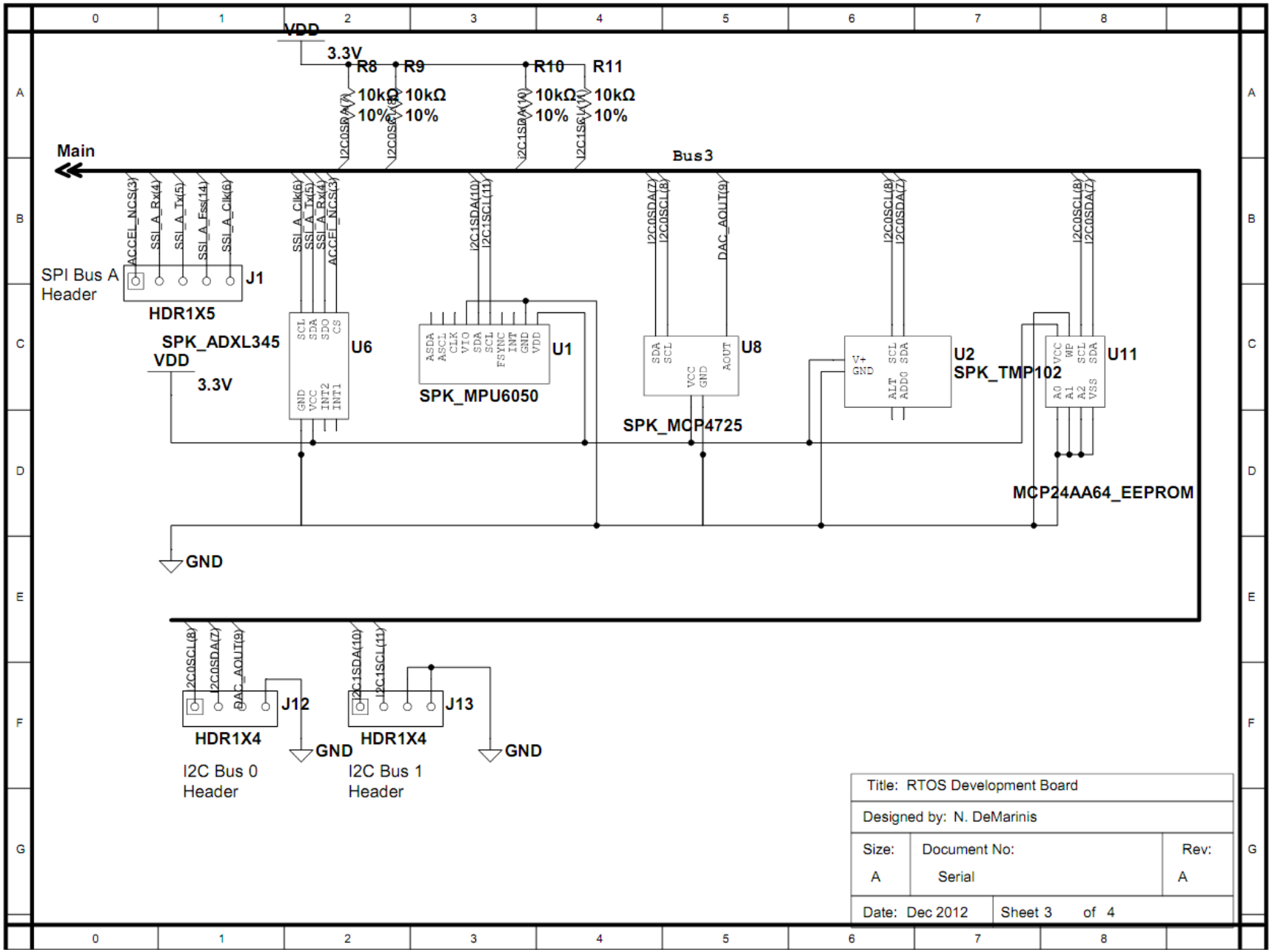
# Appendix C. Peripheral Board Bill of Materials

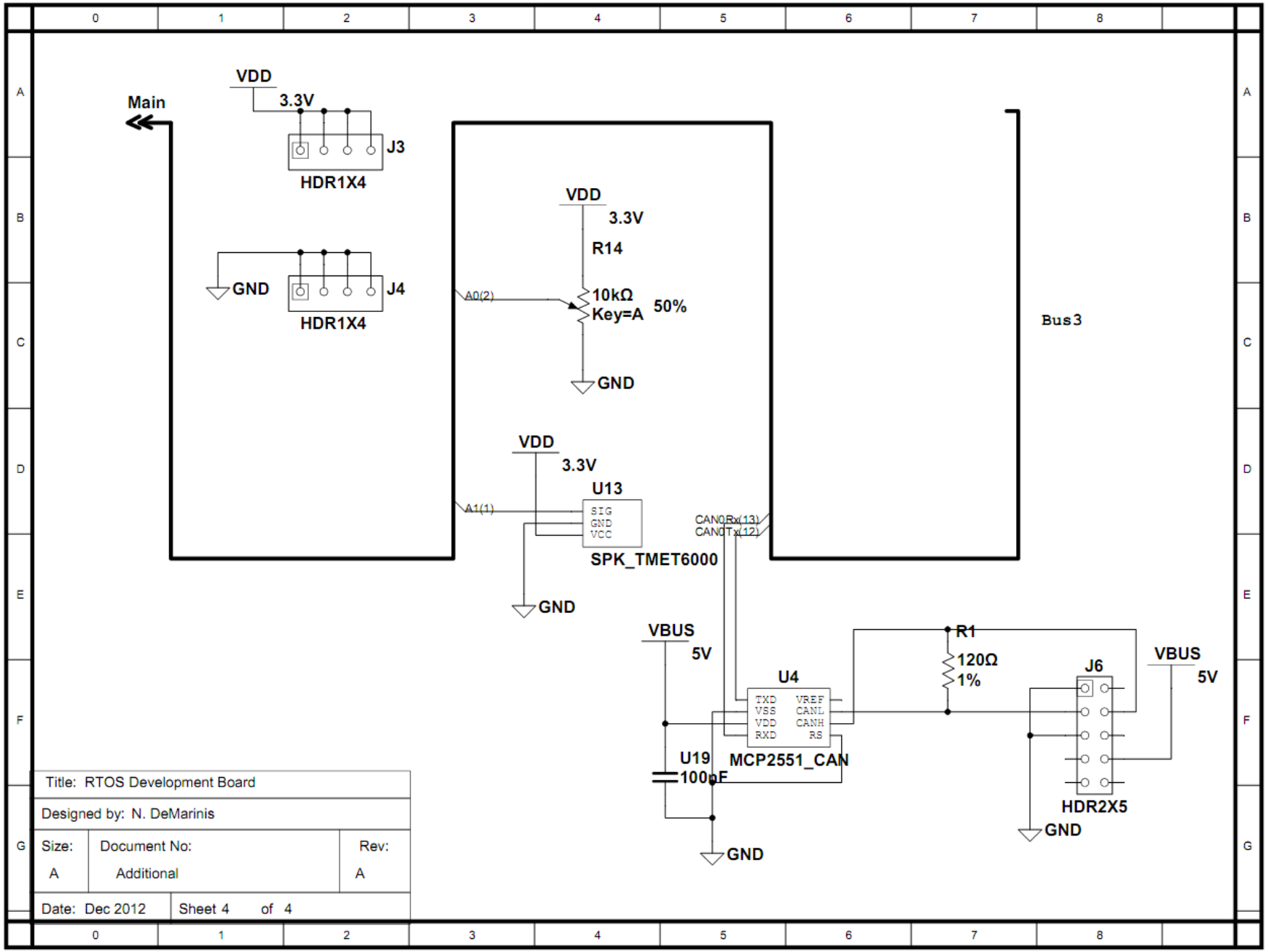
| RTOS Development Board MQP | | | | | |
|---|---|---|---|---|---|
| **Bill of Materials** | | **Created by: N. DeMarinis** | | **4/22/2013** | **Rev B** |
| **Microcontroller Board** | | | | | |
| **Vendor** | **Part No** | **Description** | **Qty** | **Unit Cost** | **Line Total** |
| Texas Instruments | EKS-LM4F232 | LM4F232 Evaluation Kit | 1 | $150.00 | $150.00 |
| **Peripherals** | | | | | |
| **Vendor** | **Part No** | **Description** | **Qty** | **Unit Cost** | **Line Total** |
| Crystalfontz | CFAF320240-035T-TS-CB | 320x240 TFT LCD Display | 1 | $86.78 | 86.78 |
| Crystalfontz | CFAH1602BNGGJTV | 2x16 Character Display | 1 | $19.48 | 19.48 |
| Sparkfun | SEN-09836 | ADXL345 Accelerometer Breakout | 1 | $27.95 | 27.95 |
| Sparkfun | SEN-11028 | MPU6050 Breakout board | 1 | $39.95 | 39.95 |
| Sparkfun | BOB-08688 | TMET6000 Ambient Light Sensor Breakout | 1 | $4.95 | 4.95 |
| Sparkfun | BOB-08736 | MCP4725 $I^2C$ DAC Breakout | 1 | $4.95 | 4.95 |
| Sparkfun | SEN-09418 | TMP102 $I^2C$ Temp Sensor Breakout | 1 | $5.95 | 5.95 |
| Digikey | GH5002-ND | Greyhill 4x3 Keypad | 1 | $14.20 | 14.2 |
| Digikey | MCP2551-I/P-ND | MCP2551 CAN Transceiver | 1 | $1.12 | 1.12 |
| Digikey | 24FC64F-I/P-ND | MCP24AA64 $I^2C$ EEPROM | 1 | $0.56 | 0.56 |
| | | | | **Total Peripherals** | **$205.89** |
| **Misc. Components** | | | | | |
| **Vendor** | **Part No** | **Description** | **Qty** | **Unit Cost** | **Line Total** |
| Any | Any | 10kΩ 10% Resistor | 4 | $0.01 | $0.04 |
| Any | Any | 120Ω 10% Resistor | 1 | $0.01 | $0.01 |
| Any | Any | 10kΩ Potentiometer | 1 | $0.05 | $0.05 |
| Any | Any | 0.01µF Capacitor | 2 | $0.01 | $0.02 |
| Digikey | 732-2860-ND | 32pos 0.1" Pitch Female Socket | 6 | $1.49 | $8.94 |
| Digikey | A33159-ND | 2x5 Male Header, Shrouded | 1 | $1.17 | $1.17 |
| Digikey | A100204-ND | 8DIP IC Socket | 2 | $0.19 | $0.38 |
| Digikey | SAM1035-50-ND | 50pos 0.1" Pitch Male header | 3 | $2.18 | $6.54 |
| | | | | **Total Misc. Components** | **$17.15** |
| | | | | **PCB Manufacturing (Min qty 4)** | **$33.00** |
| **Grand Total** | | | | | **$406.04** |

# Appendix D. Peripheral Board Schematic, Rev A

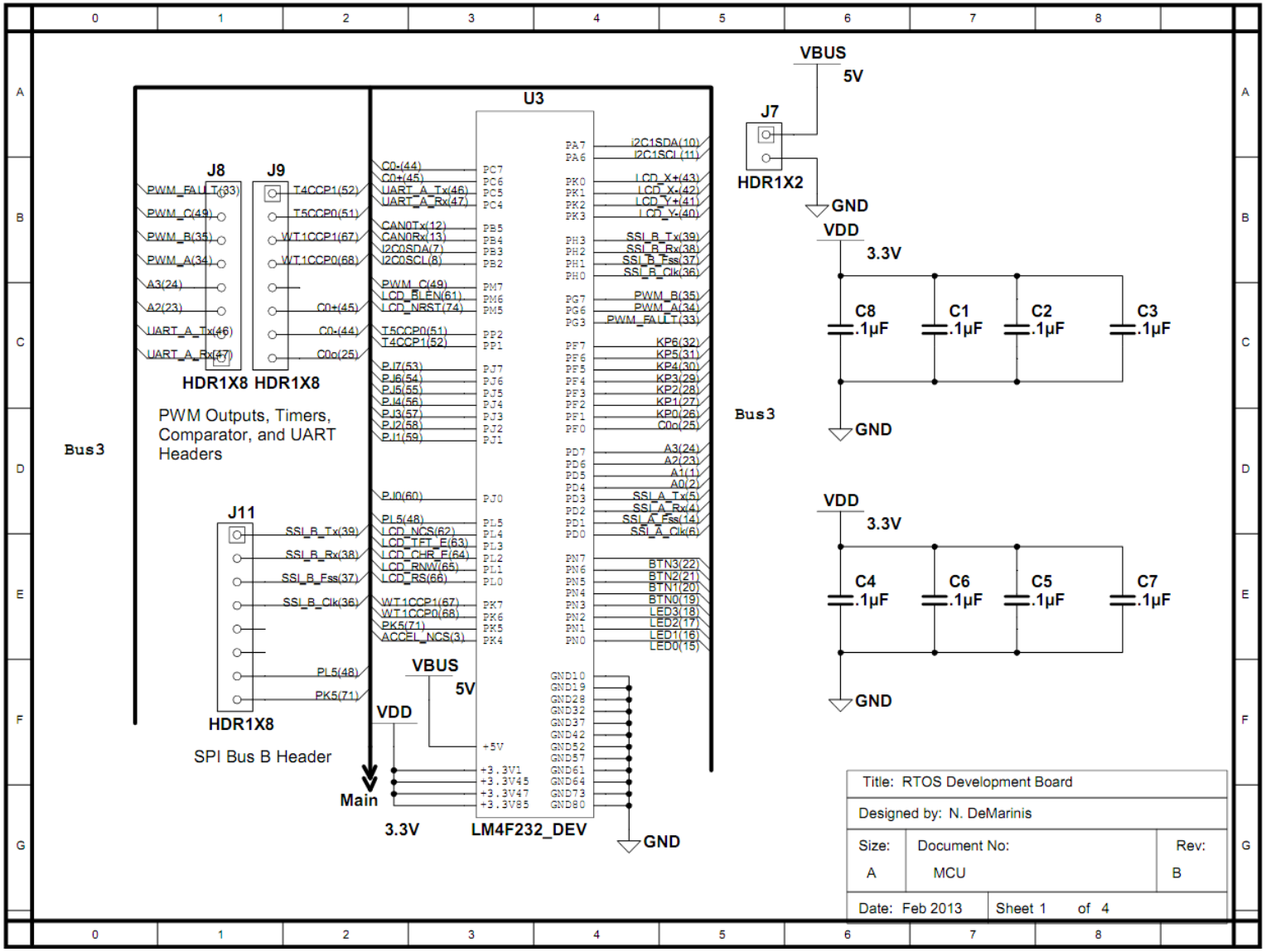


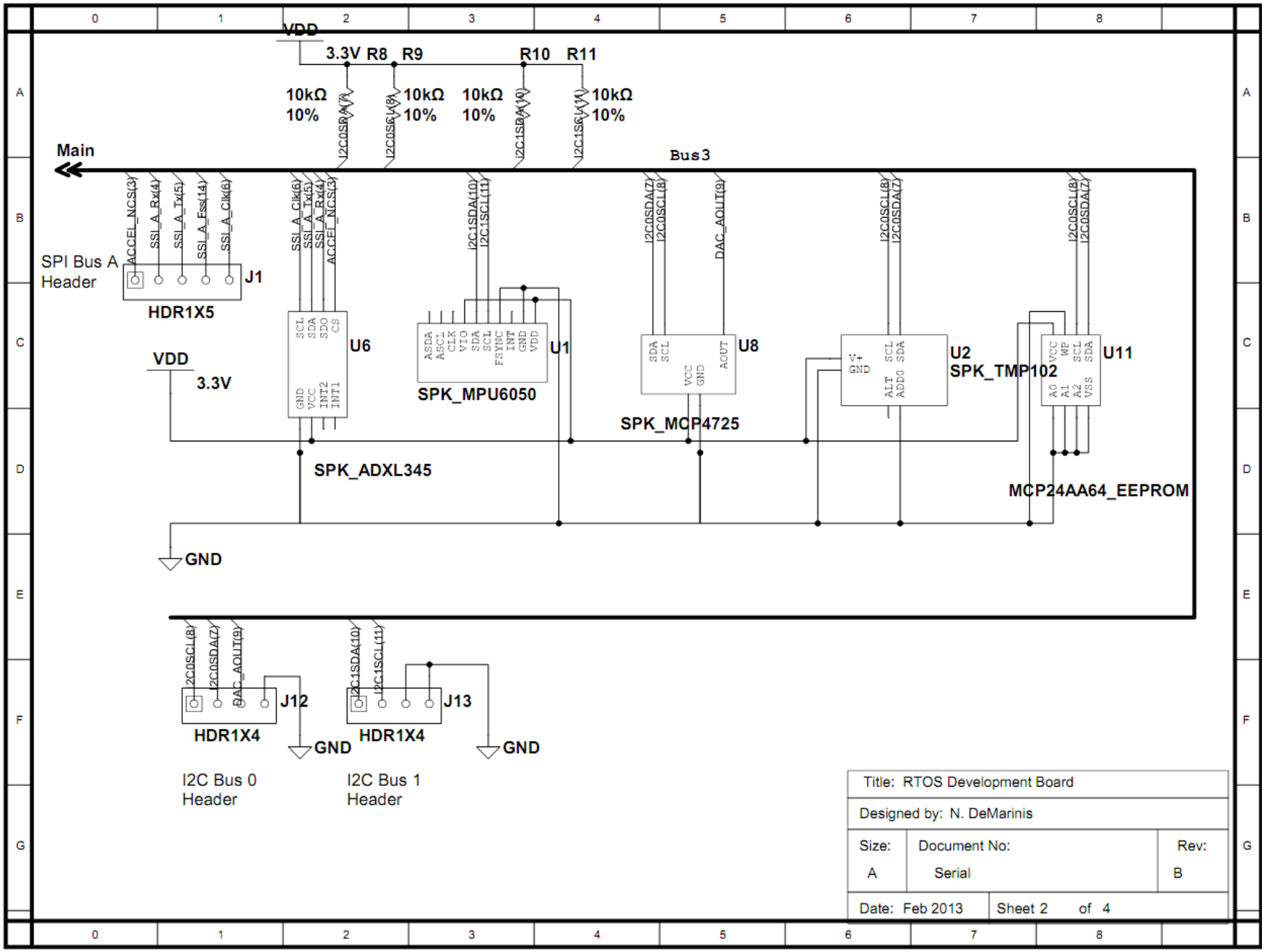

Main
Bus3
KP0(26)
KP1(27)
KP2(28)
KP3(30)
KP4(31)
KP5(32)
KP6(33)
C1 C2 C3
R1 R2 R3 R4
1 2 3
4 5 6
7 8 9
* 0 #
U15
4x3_Keypad
LCD_CHR_E(65)
LCD_TFT_E(64)
LCD_RNW(66)
LCD_RS(67)
LCD_BLEN(62)
NRST(51)
LCD_NCS(63)
J5
HDR1X8
Display Control Signals
LCD_X-(43)
LCD_X+(44)
PJ7(54)
PJ6(55)
PJ5(56)
PJ4(57)
PJ3(58)
PJ2(59)
PJ1(60)
PJ0(61)
J2
LCD_Y+(42)
LCD_Y-(41)
HDR2X10
Parallel Bus & Touchscreen
VDD
3.3V
U16
LCD_BLEN(62)
NRST(51)
LCD_TFT_E(64)
LCD_NCS(63)
LCD_RS(67)
LCD_RNW(66)
VDD
3.3V
PS3
PS2
PS1
PS0
PJ7(54)
PJ6(55)
PJ5(56)
PJ4(57)
PJ3(58)
PJ2(59)
PJ1(60)
PJ0(61)
DB17
DB16
DB15
DB14
DB13
DB12
DB11
DB10
DB9
DB8
DB7
DB6
DB5
DB4
DB3
DB2
DB1
DB0
D_EN
HSYNC
VSYNC
DCLK
Y+
X+
Y-
X-
NC
LCD_X-(43)
LCD_Y-(41)
LCD_X+(44)
LCD_Y+(42)
50%
CFAF_320_240_LCD_CB
GND
Bus3
PJ7(54)
PJ6(55)
PJ5(56)
PJ4(57)
PJ3(58)
PJ2(59)
PJ1(60)
PJ0(61)
LCD_CHR_E(65)
LCD_RNW(66)
LCD_RS(67)
VDD
3.3V
R12
10kΩ
Key=A
NC16 VEE D7 D6 D5 D4 D3 D2 D1 D0 E R/NW RS Vo VDD VSS
CFAF_1602_CHAR_LCD
U18
GND
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: Parallel
Rev: A
Date: Dec 2012
Sheet 2 of 4

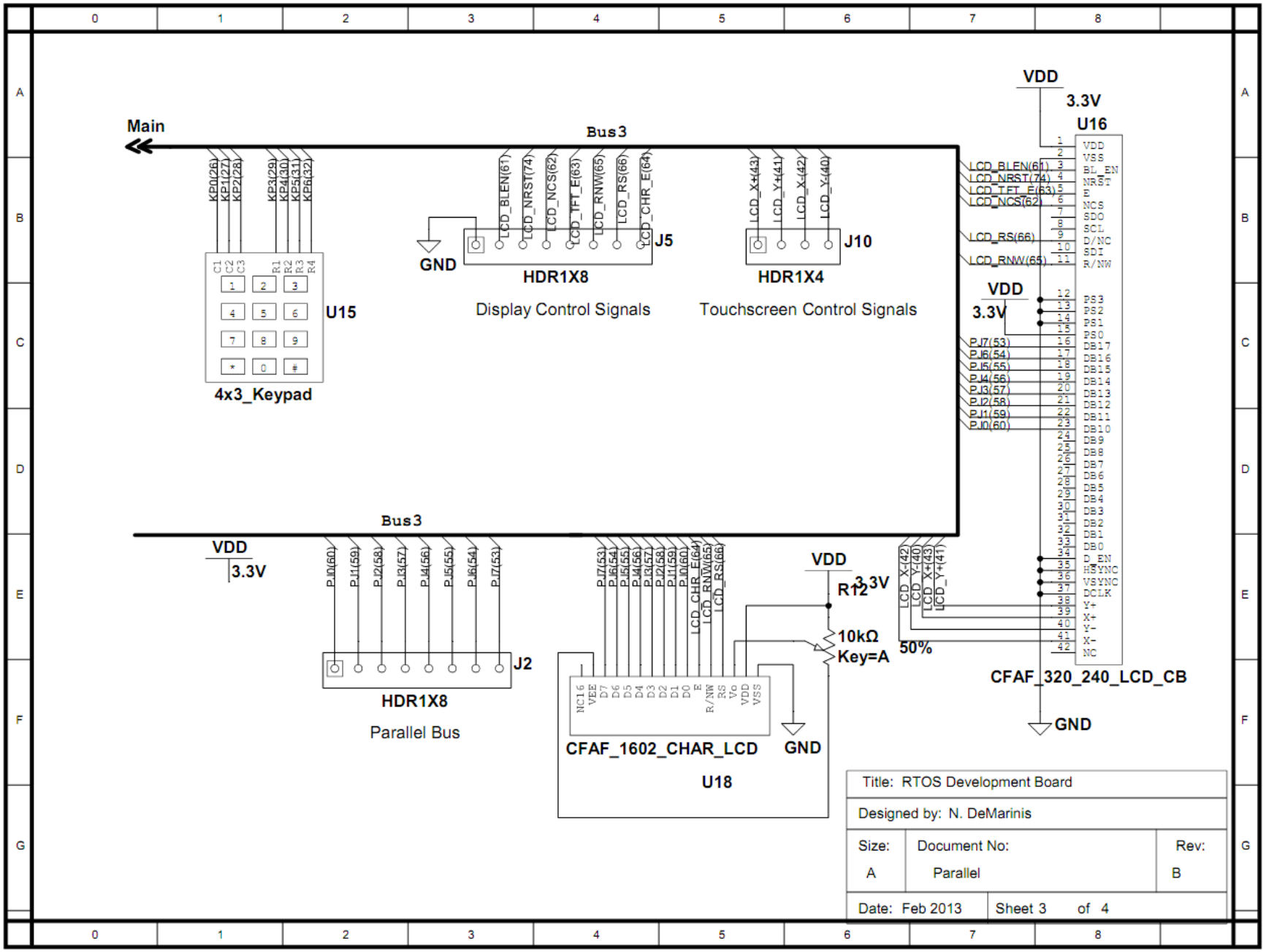

VDD
3.3V
R8
R9
R10
R11
10kΩ 10%
Main
Bus3
SPI Bus A Header
J1
HDR1X5
SPK_ADXL345
VDD
3.3V
U6
SPK_MPU6050
U1
SPK_MCP4725
U8
U2
SPK_TMP102
U11
MCP24AA64_EEPROM
GND
J12
HDR1X4
I2C Bus 0 Header
J13
HDR1X4
I2C Bus 1 Header
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: Serial
Rev: A
Date: Dec 2012
Sheet 3 of 4

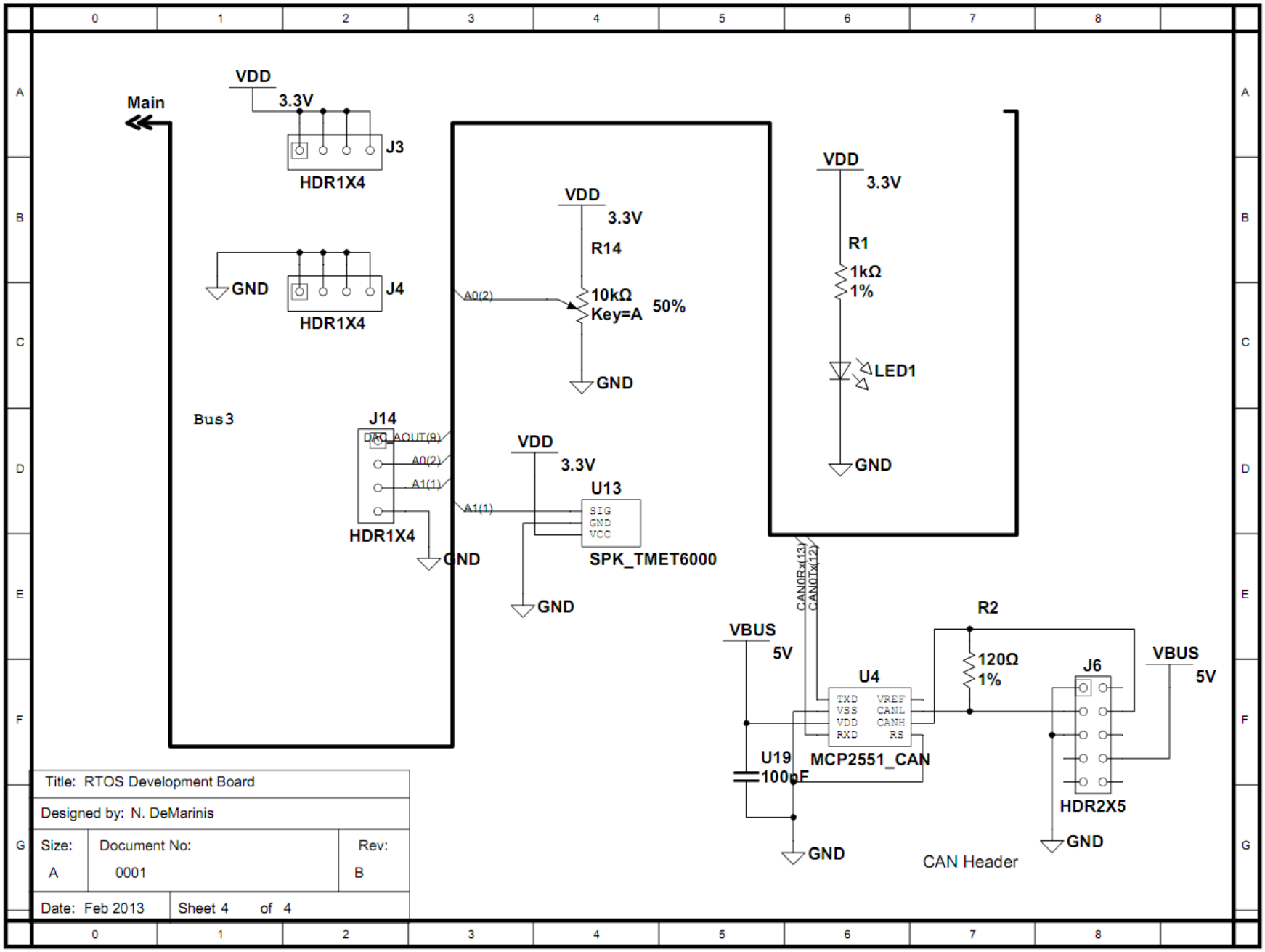

Main
VDD
3.3V
J3
HDR1X4
GND
J4
HDR1X4
VDD
3.3V
R14
A0(2)
10kΩ
Key=A
50%
GND
Bus3
VDD
3.3V
U13
A1(1)
SIG
GND
VCC
SPK_TMET6000
GND
CAN0Rx(13)
CAN0Tx(12)
VBUS
5V
U4
TXD
VSS
VDD
RXD
VREF
CANL
CANH
RS
MCP2551_CAN
U19
100pF
GND
R1
120Ω
1%
J6
HDR2X5
GND
VBUS
5V
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: Additional
Rev: A
Date: Dec 2012
Sheet 4 of 4

# Appendix E. Peripheral Board Schematic, Rev B



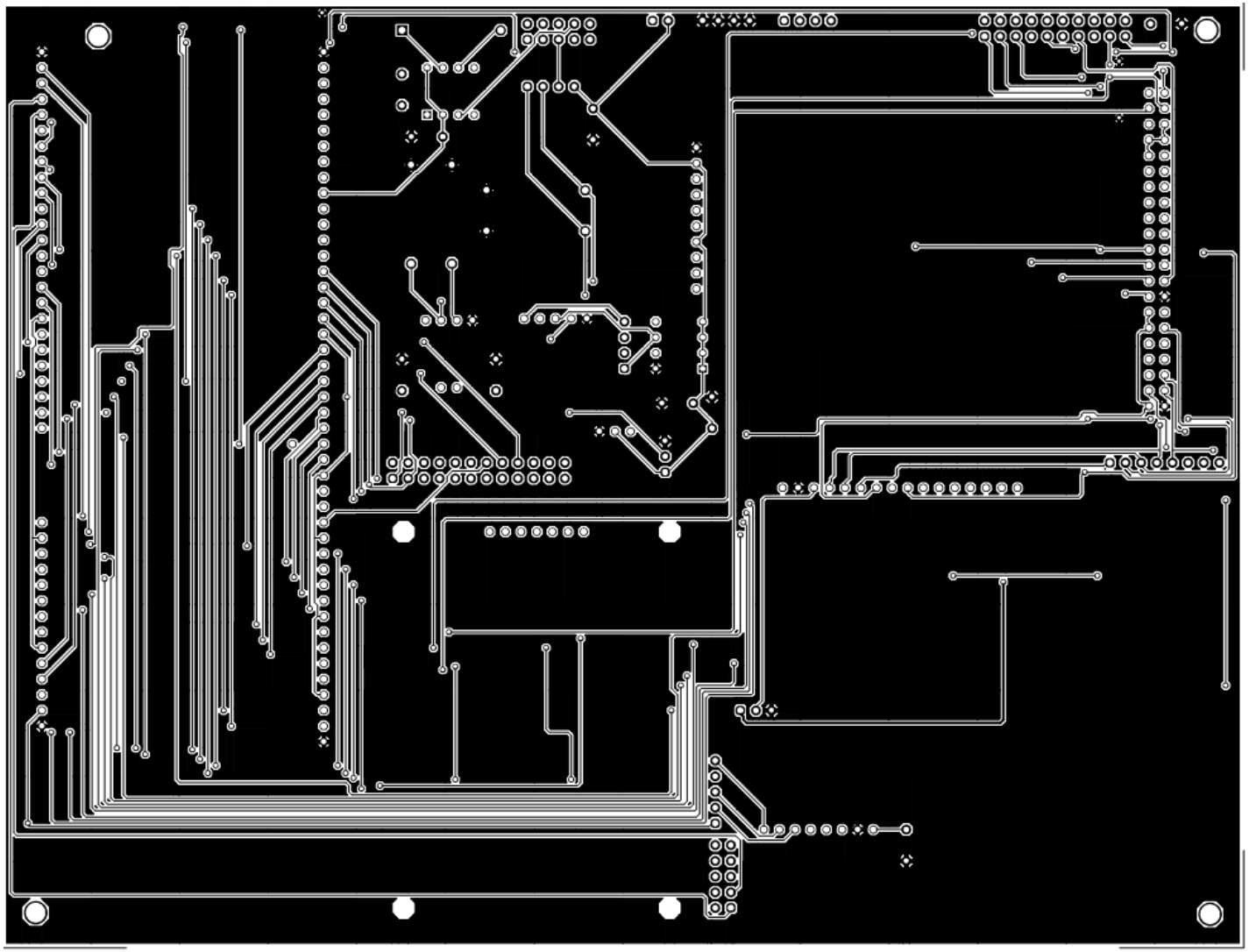

VDD
3.3V
R8
R9
R10
R11
10kΩ 10%
10kΩ 10%
10kΩ 10%
10kΩ 10%
Main
Bus3
SPI Bus A Header
J1
HDR1X5
U6
SPK_ADXL345
U1
SPK_MPU6050
U8
SPK_MCP4725
U2
SPK_TMP102
U11
MCP24AA64_EEPROM
GND
J12
HDR1X4
J13
HDR1X4
I2C Bus 0 Header
I2C Bus 1 Header
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: Serial
Rev: B
Date: Feb 2013
Sheet 2 of 4


Main
Bus3
KP0(26)
KP1(27)
KP2(28)
KP3(29)
KP4(30)
KP5(31)
KP6(32)
C1 C2 C3
R1 R2 R3 R4
1 2 3
4 5 6
7 8 9
* 0 #
U15
4x3_Keypad
GND
LCD_BLEN(61)
LCD_NRST(74)
LCD_NCS(62)
LCD_TFT_E(63)
LCD_RNW(65)
LCD_RS(66)
LCD_CHR_E(64)
J5
HDR1X8
Display Control Signals
LCD_X+(43)
LCD_Y+(41)
LCD_X-(42)
LCD_Y-(40)
J10
HDR1X4
Touchscreen Control Signals
VDD
3.3V
U16
LCD_BLEN(61)
LCD_NRST(74)
LCD_TFT_E(63)
LCD_NCS(62)
LCD_RS(66)
LCD_RNW(65)
PJ7(53)
PJ6(54)
PJ5(55)
PJ4(56)
PJ3(57)
PJ2(58)
PJ1(59)
PJ0(60)
LCD_X-(42)
LCD_Y-(40)
LCD_X+(43)
LCD_Y+(41)
CFAF_320_240_LCD_CB
Bus3
VDD
3.3V
PJ0(60)
PJ1(59)
PJ2(58)
PJ3(57)
PJ4(56)
PJ5(55)
PJ6(54)
PJ7(53)
J2
HDR1X8
Parallel Bus
PJ7(53)
PJ6(54)
PJ5(55)
PJ4(56)
PJ3(57)
PJ2(58)
PJ1(59)
PJ0(60)
LCD_CHR_E(64)
LCD_RNW(65)
LCD_RS(66)
NC16 VEE D7 D6 D5 D4 D3 D2 D1 D0 E R/NW RS Vo VDD VSS
CFAF_1602_CHAR_LCD
U18
VDD
3.3V
R12
10kΩ
Key=A
50%
GND
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: Parallel
Rev: B
Date: Feb 2013
Sheet 3 of 4

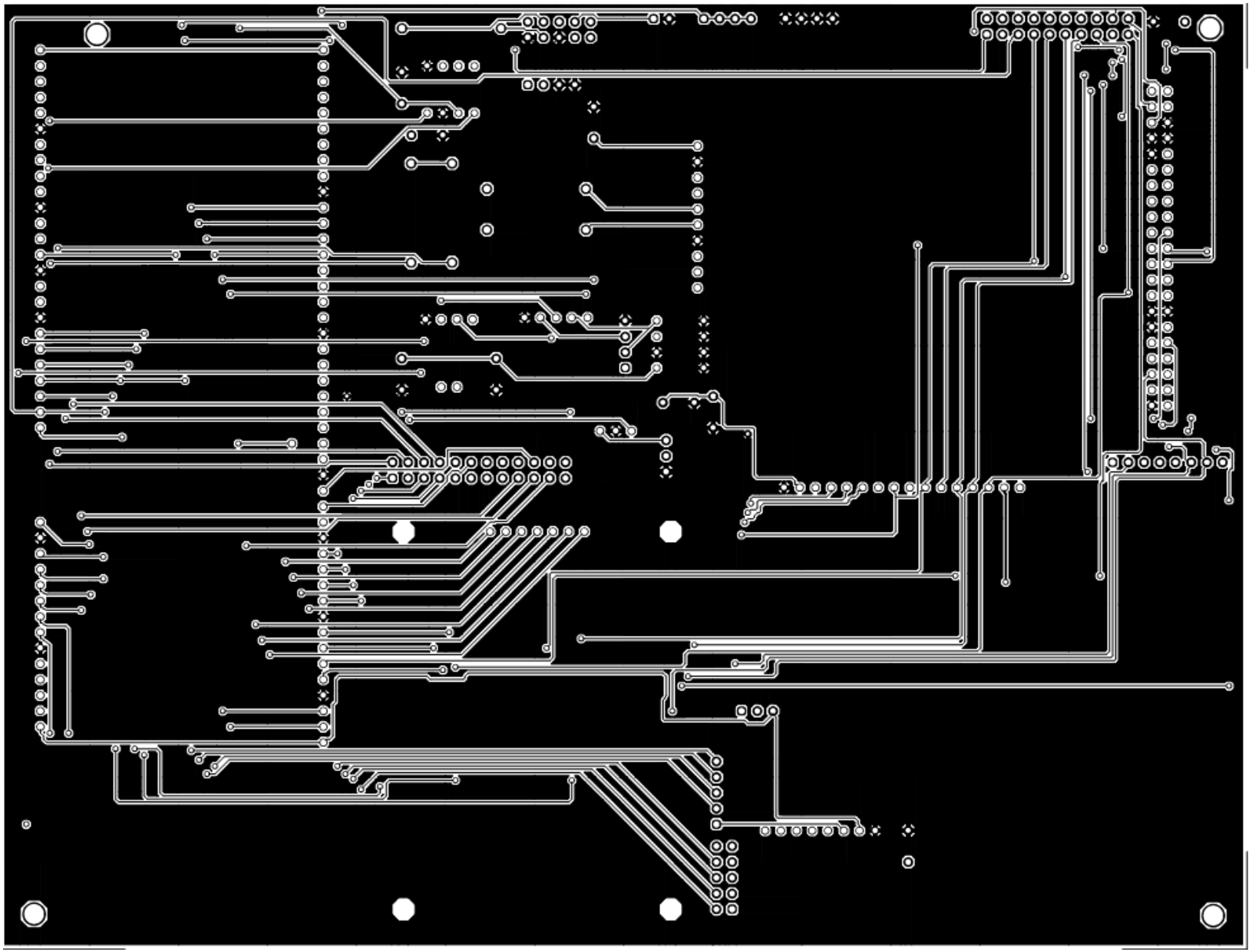

Main
VDD
3.3V
J3
HDR1X4
GND
J4
HDR1X4
Bus3
VDD
3.3V
R14
10kΩ
Key=A
50%
A0(2)
GND
J14
DAC_AOUT(9)
A0(2)
A1(1)
HDR1X4
GND
VDD
3.3V
U13
SIG
GND
VCC
A1(1)
SPK_TMET6000
GND
VDD
3.3V
R1
1kΩ
1%
LED1
GND
CAN0Rx(13)
CAN0Tx(12)
R2
120Ω
1%
VBUS
5V
U4
TXD
VSS
VDD
RXD
VREF
CANL
CANH
RS
MCP2551_CAN
U19
100pF
GND
J6
HDR2X5
GND
VBUS
5V
CAN Header
Title: RTOS Development Board
Designed by: N. DeMarinis
Size: A
Document No: 0001
Rev: B
Date: Feb 2013
Sheet 4 of 4

# Appendix F. Peripheral Board PCB Layout, Rev A

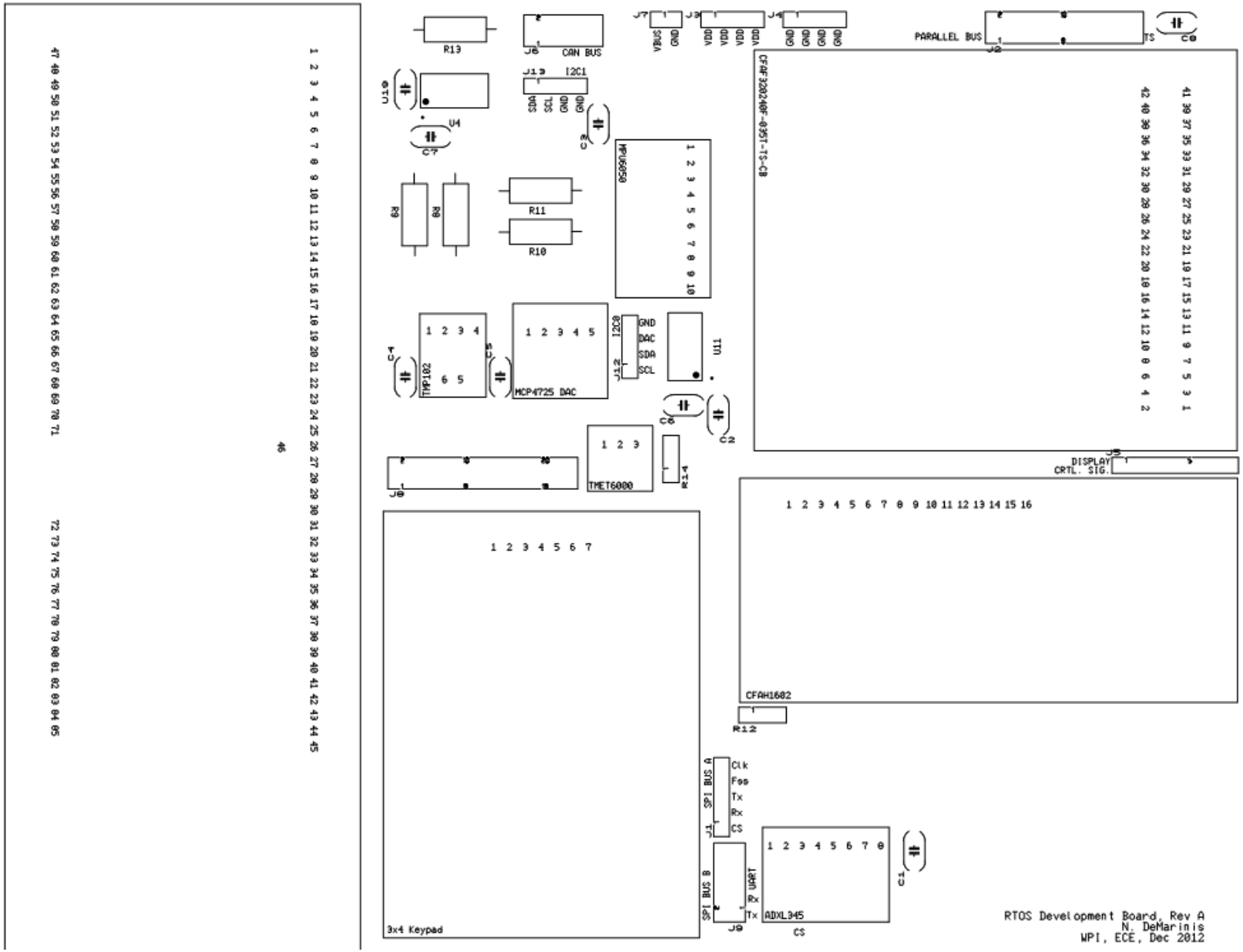
Top Copper Layer



## Bottom Copper Layer

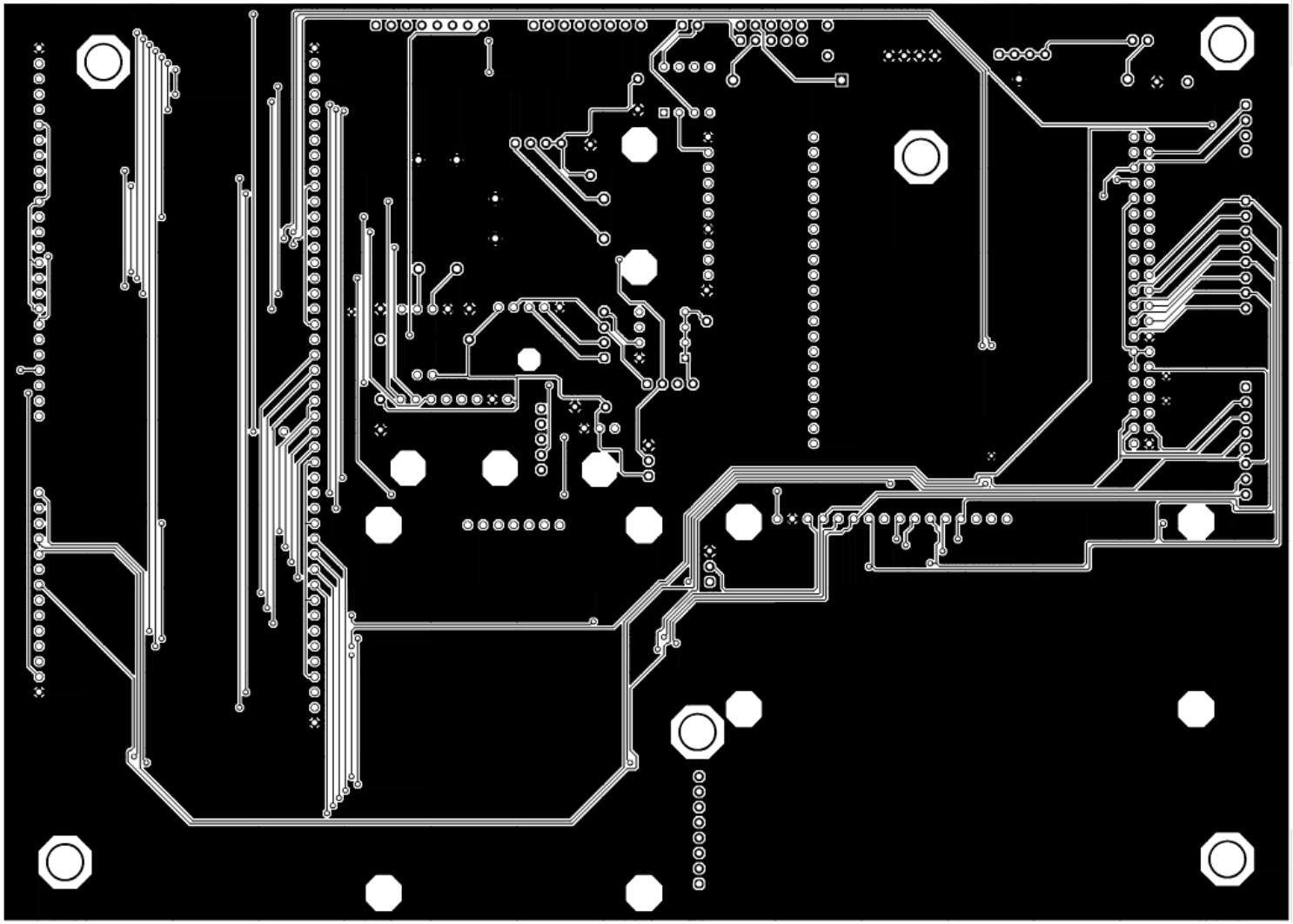


## Top Silkscreen Layer

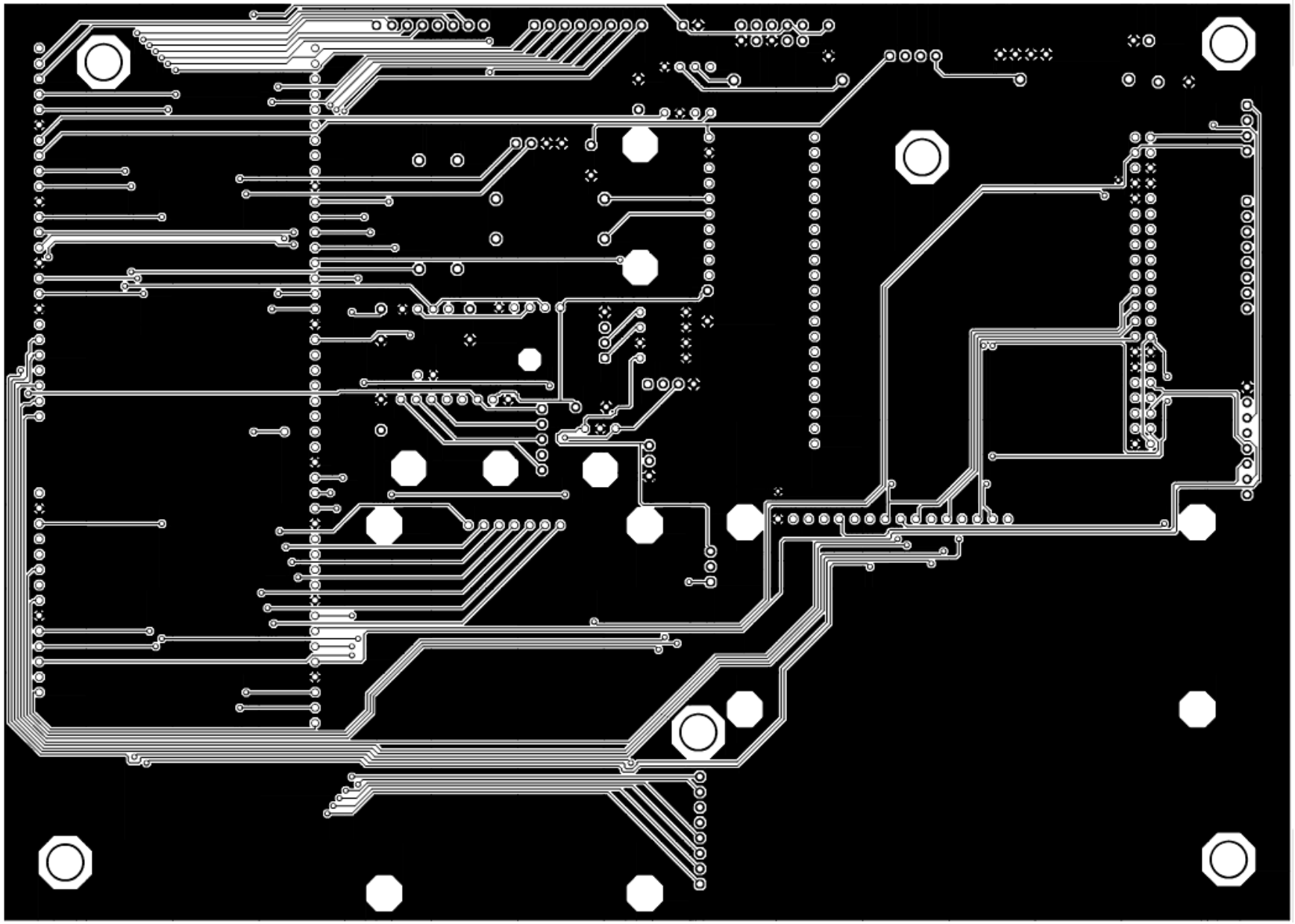


# Appendix G. Peripheral Board PCB Layout, Rev B

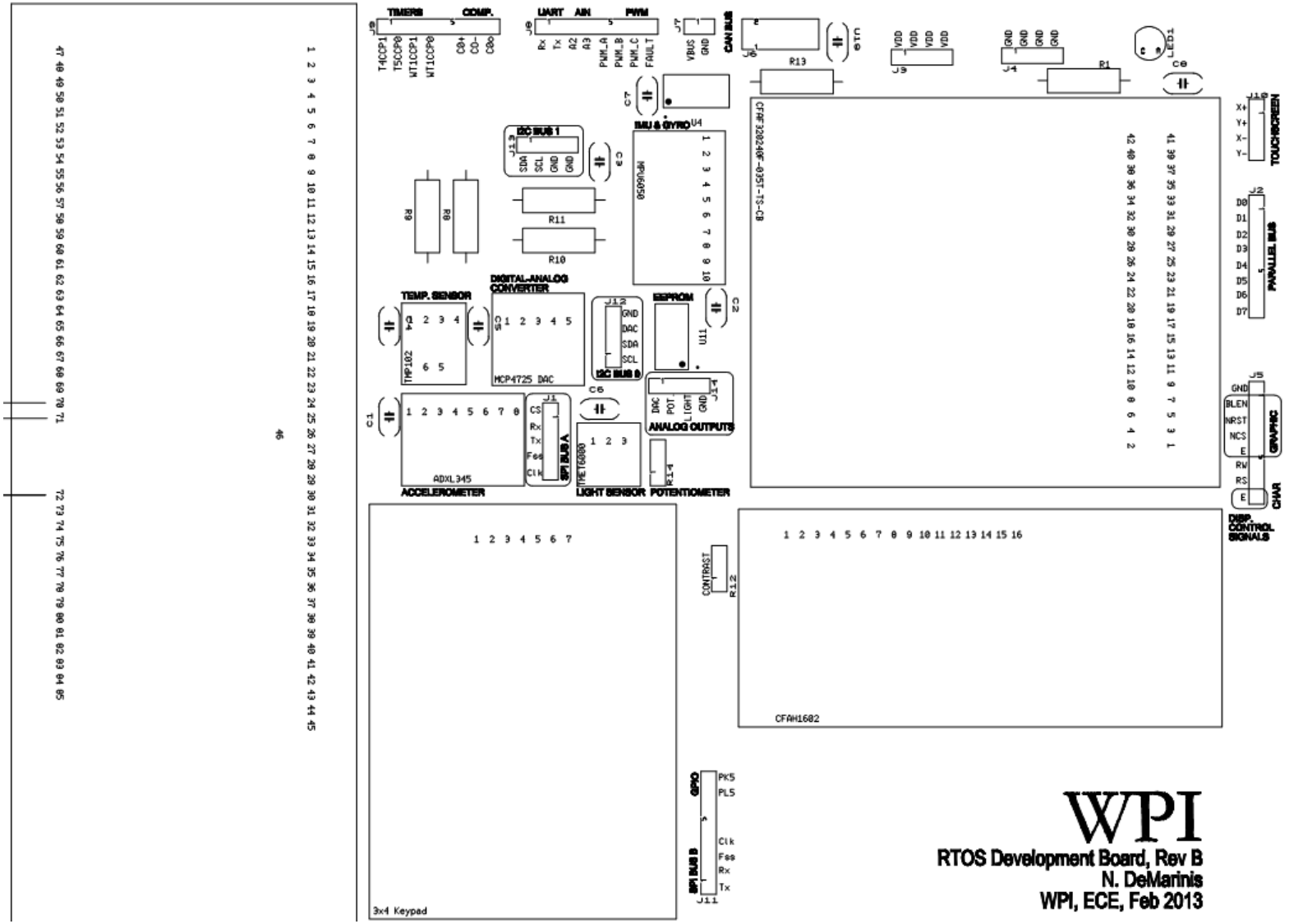
Top Copper Layer



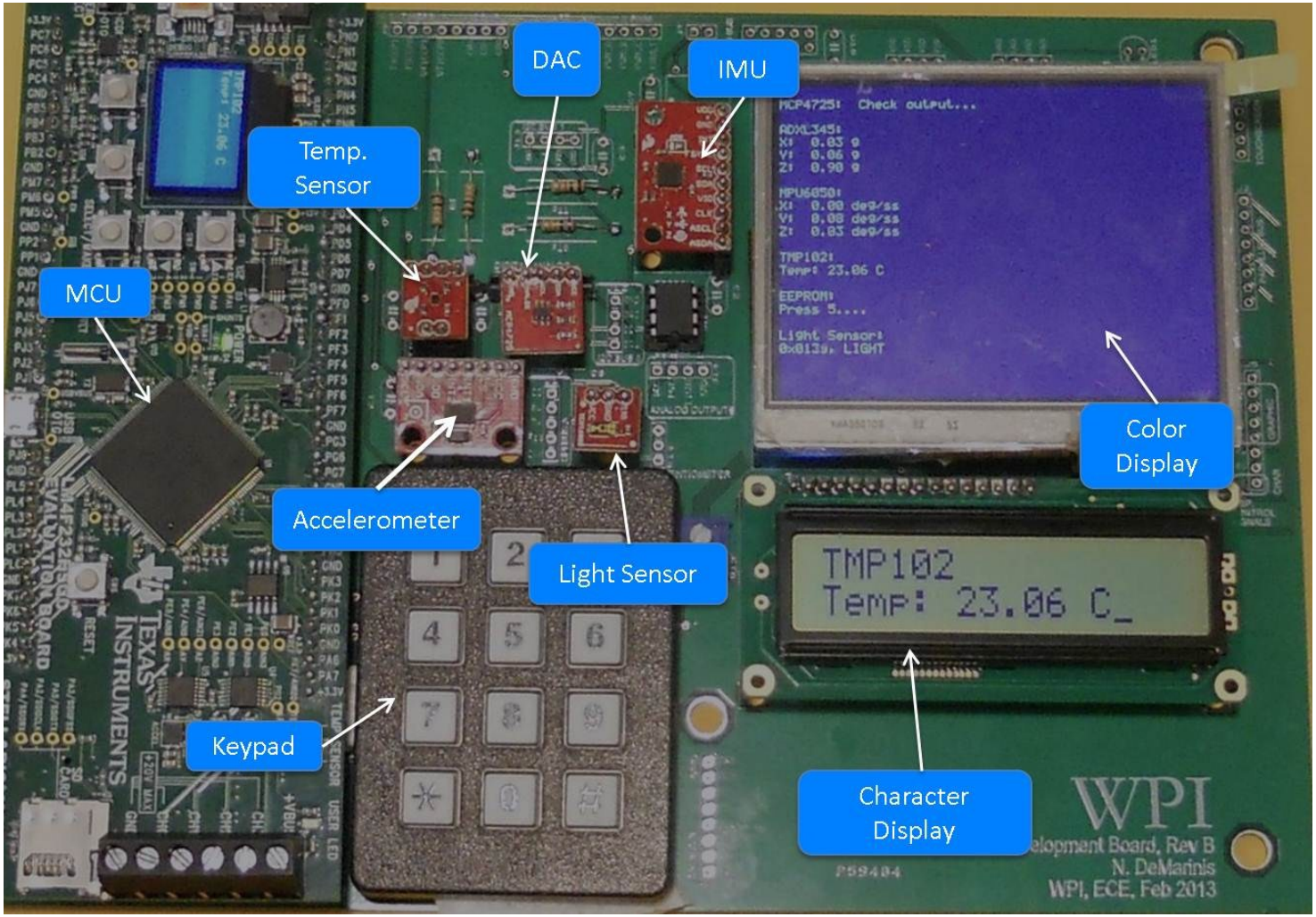
## Bottom Copper Layer



## Top Silkscreen Layer



## Appendix H. Peripheral Board with photograph with labeled components



# Appendix I. Excerpt from ECE 3849 Lab

**Step 2: ADC Sampling**

Your first challenge is to acquire ADC samples at 500 kSamples/sec without missing any. To configure the ADC, you may use the following incomplete code as a starting point:

```
SysCtlPeripheralEnable(...); // enable the ADC
SysCtlADCSpeedSet(...);      // specify 500ksps
ADCSequenceDisable(...);     // choose ADC sequence 0; disable before configuring
ADCSequenceConfigure(...);   // specify the "Always" trigger
ADCSequenceStepConfigure(...); // in the 0th step, sample channel 0
                             // enable interrupt, and make it the end of sequence
ADCIntEnable(...);           // enable ADC interrupt from sequence 0
ADCSequenceEnable(...);      // enable the sequence.  it is now sampling
IntPrioritySet(...);        // set ADC interrupt priority in the interrupt controller
IntEnable(...);              // enable ADC interrupt
```

Consult the ADC chapter of Stellaris Peripheral Driver Library User's Guide to fill in the arguments of the driver function calls. The TI ADC Application Note can also come in useful. You will also need to include the appropriate header for the ADC driver.

We have just configured the ADC to acquire a sample and generate an interrupt every 2 μs. The ADC ISR must process this sample before the next one is acquired. This gives the ADC ISR only 100 CPU cycles to complete its work (remember, this includes the latency). This is an extremely tight timing requirement, but the Cortex-M3 turns out to be up to the challenge. We will need to perform some optimization to meet this goal. Later on we will learn how to relax this timing constraint (but not in this lab). The following is the recommended structure of the ADC ISR:

```
#define ADC_BUFFER_SIZE 2048 // must be a power of 2
#define ADC_BUFFER_WRAP(i) ((i) & (ADC_BUFFER_SIZE - 1)) // index wrapping macro
volatile int g_iADCBufferIndex = ADC_BUFFER_SIZE - 1;    // latest sample index
volatile unsigned short g_pusADCBuffer[ADC_BUFFER_SIZE]; // circular buffer
volatile unsigned long g_ulADCErrors = 0; // number of missed ADC deadlines

void ADC_ISR(void)
{
    <...>; // clear ADC sequence0 interrupt flag in the ADCISC register
    if (ADC0_OSTAT_R & ADC_OSTAT_OV0) { // check for ADC FIFO overflow
        g_ulADCErrors++;    // count errors - step 1 of the signoff
        ADC0_OSTAT_R = ADC_OSTAT_OV0; // clear overflow condition
    }
    int buffer_index = ADC_BUFFER_WRAP(g_iADCBufferIndex + 1);
    g_pusADCBuffer[buffer_index] = <...>; // read sample from the ADC sequence0 FIFO
    g_iADCBufferIndex = buffer_index;
}
```

The ADC ISR is very simple:

1. Acknowledge the ADC interrupt (so that it will not interrupt again on return).
2. Detect if a deadline was missed by checking the overflow flag of the ADC hardware.
3. Read a sample from the ADC and store in a buffer array.

Although there is nothing particularly difficult about this code, enough students had extreme problems writing it themselves in the past, so the code is now mostly given away. Your only tasks are to find how to clear the interrupt flag that originally caused this ISR to be called, and to read in the actual ADC sample. You have to do this using **direct register access**, not driver function calls. Recall the explanation in Lab 0 and Chapter 2 of the Stellaris Peripheral Driver Library User's Guide. In this case, the overhead associated with driver function calls is enough to start missing deadlines (you can verify this).

Make sure to refer to this ISR in the appropriate interrupt vector in startup_ccs.c. At this point, you can run your code and verify that the ISR is being called and the ADC error counter stays at zero (it should increment if you pause your program for debugging). If the counter is steadily incrementing, try to find the cause. One possibility is incorrect interrupt priority assignment (if you still have the Lab 0 timer ISR running). If your oscillator and voltage divider are connected, you should be able to observe the sampled waveform in `g_pusADCBuffer`.

# Appendix J. Embedded Systems Cookbook: A User Guide for ECE 3849

## Overview

The purpose of this guide is to demonstrate how to use the integrated development environment (IDE) and peripherals for embedded software development in ECE3849. This document will also introduce you to the basics of working with TI's Code Composer Studio (CCS) environment and software libraries that will be used throughout the course.

**If you are new to ECE3849, read this guide first.**

## Required software for development

To develop software for ECE3849, your system requires two pieces of software:

### TI Code Composer Studio (CCS)

CCS is an integrated development environment (IDE), which provides functionality for editing, compiling, and debugging code on the development boards for ECE3849. CCS is based on the Eclipse IDE, but uses a proprietary compiler and debugger built by TI for their microprocessors.

The lab machines in AK113 have been preinstalled with CCS version 5.2.1, so you do not need to install it. If you want to install it on your own computer, you can download it from http://processors.wiki.ti.com/index.php/Download_CCS. See **Installing CCS on your own computer** for details.

### StellarisWare software library

StellarisWare is a library of useful code built by TI for interfacing with the on-chip peripherals (such as the interrupt controller or the SPI interface) for the Stellaris line of microcontrollers we will be using in lab. The library also provides code for some of the peripherals on TI's development boards, example code for using the peripherals, and helpful documentation.

Like CCS, StellarisWare has been preinstalled on the lab machines in AK 113. You can view the files by navigating to **C:\StellarisWare**. **Do this now to verify the software is present.**

Note the folders labeled "examples" and "docs," which contain the example code and library documentation, respectively. These will prove useful in your labs.

### Starting CCS for the first time

This section will explain the steps to configure CCS when you start it for the first time:

1. To launch CCS, click on the icon labeled "Code Composer Studio v5" on the desktop or find it in the Start menu under "All Programs > Texas Instruments", as shown in Figure 24 below.

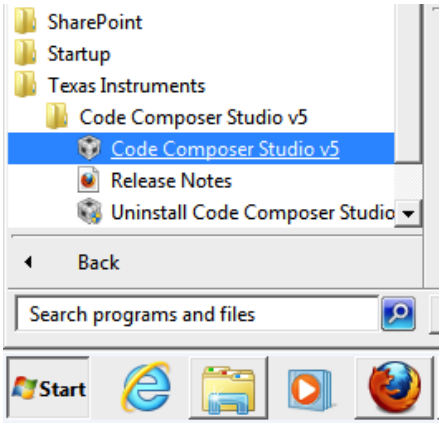



Figure 24: Launching CCS from the Start Menu

2. When CCS starts up, it will ask you to select a workspace. A **workspace** is CCS's location for storing all of your projects and configuration files.
   Create an empty folder on your shared drive (**M:**) or a flash drive for this purpose and set it as the path to the workspace. Using either your M drive or a flash drive will ensure you can access your workspace from any lab computer, not just the one you are currently using.
   Once you have set a path, your window should look like Figure 25 below.

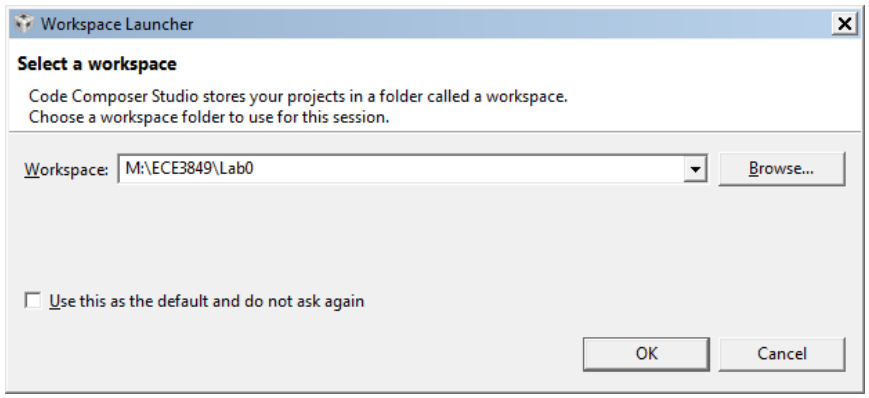



Figure 25: Example workspace prompt

**Note**: Do NOT start your path with a share name like **\\ece-homes.ece.wpi.edu\username**, which may appear as the default. This will cause issues when you try to compile your code.

3. When CCS finally starts up, it will display a prompt asking you to enter a license. Fortunately, CCS is free to use with TI's development kits like the one we use in lab. Select the free license as shown in Figure 26 below.

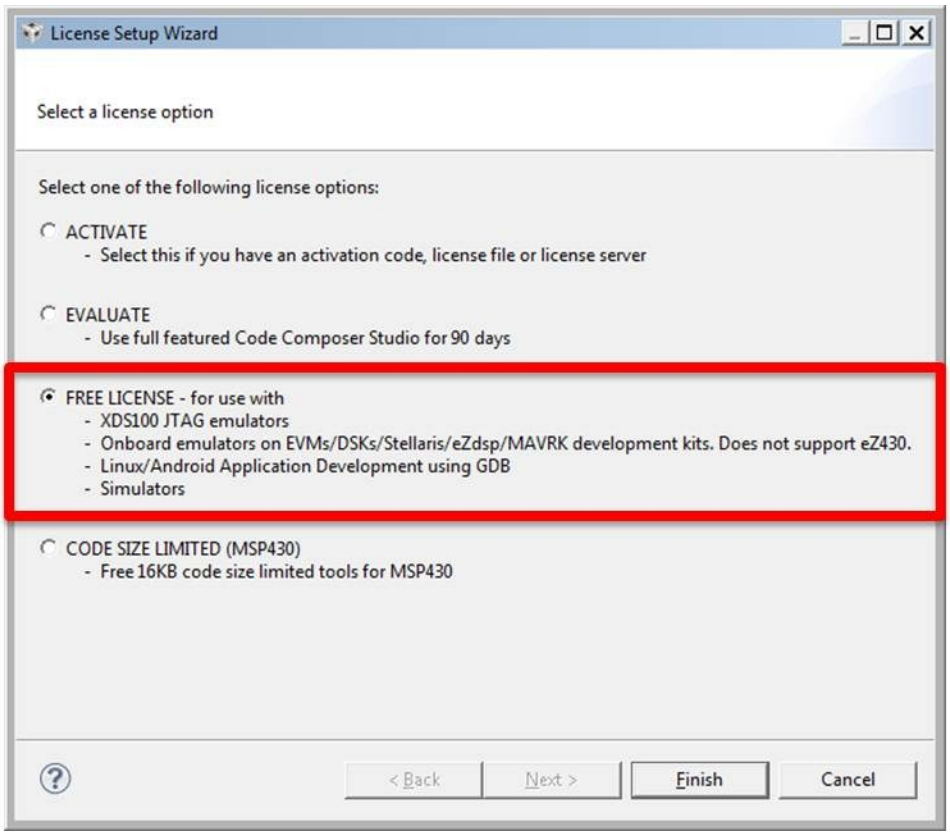



**Figure 26: CCS license prompt**

4. Following this, CCS should have finished loading and should be launched. You can now start developing or import example projects into CCS to test your hardware.

## Importing example code into CCS

TI provides example code projects in order to demonstrate the functionality on their evaluation boards. As our lab setup is built using one of these evaluation boards, you can use these as example code or to demonstrate that your hardware is working properly.

To load the example projects into your workspace:

1. From the menu bar, go to **File > Import…**
   This will display a dialog box asking you to select from a number of places from which you can import files.
2. Select **Code Composer Studio > Existing CCS Eclipse projects**, as shown in Figure 27.

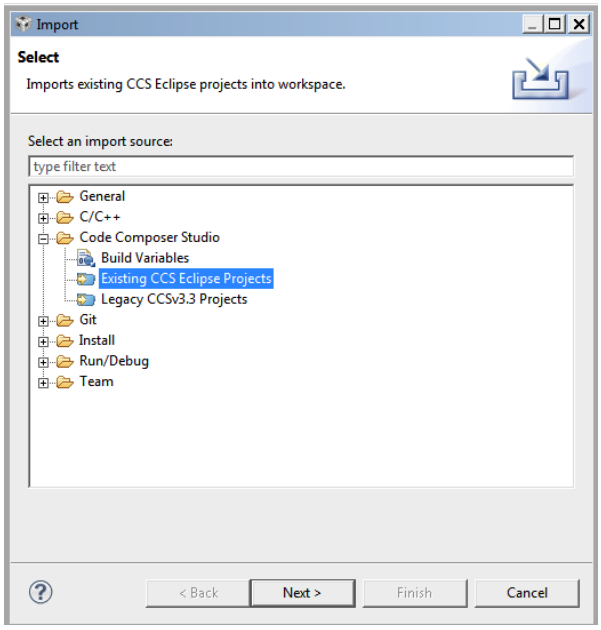



Figure 27: CCS Import Window

3. Select "Select search-directory" and click its **Browse…** button.
4. Navigate to the path **C:\StellarisWare\boards\ek-lm4f232**
   This is the location in the StellarisWare library that contains TI's code for our evaluation board, referred to as EK-LM4F232. If you want to import code from another source, you could enter its path here instead.
5. You should now see a list of projects contained in this directory that you can import. You can select as many as you wish, but it is recommended that you select at least "timers," "blinky," "hello," and "interrupts," as these provide some useful example code.
6. When you are done, click **Finish**.
7. CCS should have now imported the projects you selected.

## Navigating CCS

CCS is based on the Eclipse IDE, which is a very powerful development environment that can be daunting to navigate at first. This section will describe some of the basic components of the IDE and how it can be used.

### Initial setup

When CCS starts for the first time, it may present a window labeled "TI Resource Explorer" that hides the rest of CCS's features. To remove this, click the "X" on its tab, as shown in Figure 28.

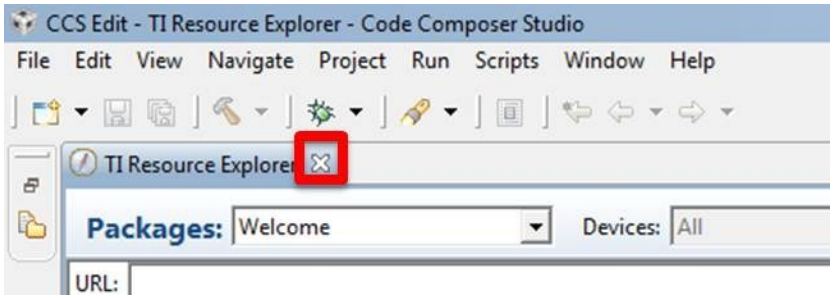



Figure 28: Closing the Resource Explorer

### Edit perspective

Eclipse and CCS group windows into different "perspectives" for different tasks. At startup, CCS will be in the "Edit" perspective, which is designed for editing and compiling your code. In the Edit perspective, CCS should have the following panes, shown in Figure 29, which are described below:

- **Project Explorer**: This pane shows all of the projects that exist in the current workspace. A **project** is a set of source code and configuration files that make up a single embedded application.
- **Editing Window**: This window displays open source files in a tabbed interface.
- **Problems Pane**: This pane lists any errors the compiler encountered when building the current project (currently empty).

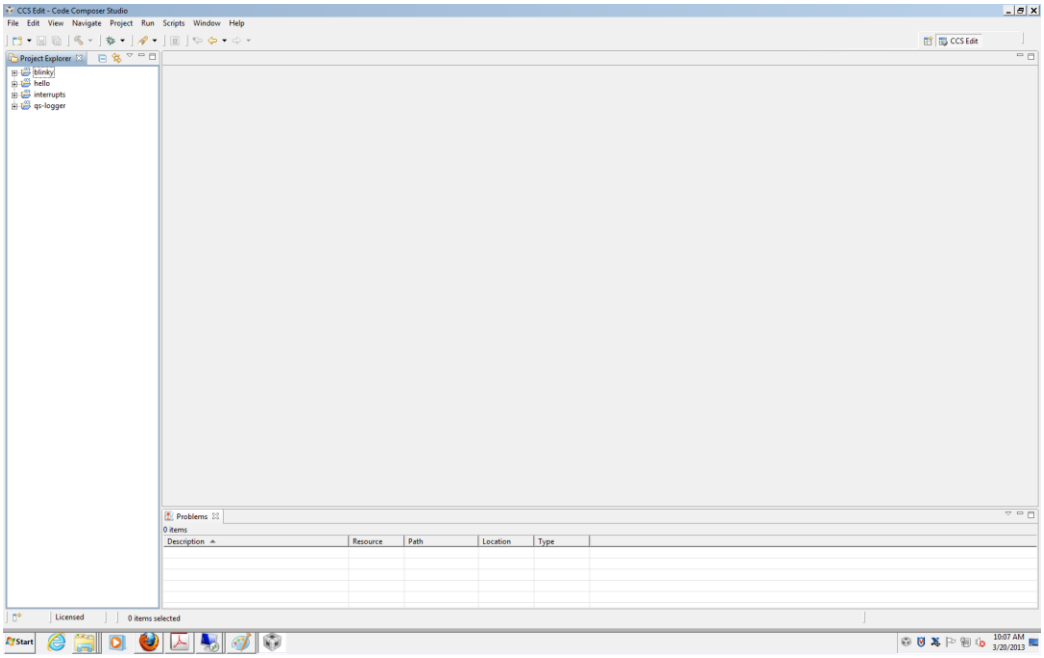



**Figure 29: CCS's Edit Perspective**

Using these panes, you can perform most of the actions you need to find files in projects, edit code, and compile it for the microcontroller. Most of the features available to do this are located in the context (right-click) menus for each project in the Project Explorer and the editing window. Some of the more useful features will be described in later sections of this guide.

## "Open" and "Closed" Projects

In Eclipse and CSS, a project in a workspace is "open" if it shows an "open folder" icon next to its name in the Project Explorer. (In Figure 30, all of the listed projects are open.) Open and closed projects behave differently in CCS:

- When a project is open, CCS will allow you to access its files, compile it, and reference files inside it.
- When a project is closed, CCS ignores the project until you open it.

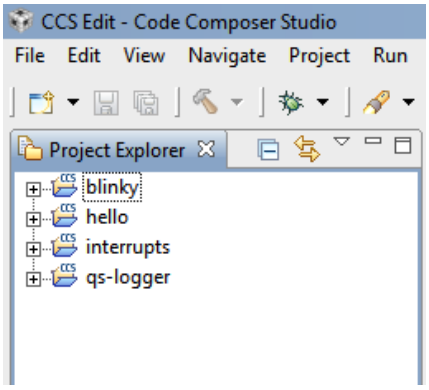



Figure 30: These four CCS projects are open

This is important because CCS (and Eclipse) can become confused if more than one project is open and try to build files across projects, which often leads to unwanted errors. In addition, it also forces CCS to scan multiple projects for certain commands, which can cause the program to run slowly. Therefore, **it is recommended that you only keep one project open at a time: the project you are currently using.**

To close an unused project:

1. Right click on the project you want to close.
2. Select Close Project.

To open an unused project:

1. Right-click on the project you want to open.
2. Select Open Project.
3. Recommended: Close any other open projects.

## Building and debugging a project

In order to run your software on the microcontroller, it must be compiled and then programmed to the chip for execution. This section describes how to compile projects and run them in CCS's debugger.

Note: This section assumes you have a working CCS example project (provided by TI) already loaded, as described in "Importing CCS projects." If you want to build and debug your own projects, see "Creating your own projects.

### Building a project

Since embedded systems can vary widely, most embedded compilers have a lot of options and require very specific configurations to work properly with your system. Fortunately, TI's example projects are already configured for our system, greatly simplifying the setup process.

To build a CCS project:

1. Right-click on the project you want to build. This tutorial will use CCS's "blinky" project, which blinks the onboard LED on the board. This is a good example project since it is very concise and easy to understand.
2. Select Build project.

This should invoke TI's compiler on the source files in the project. You should notice the Console window become active in CCS's bottom pane as this happens.

When it completes, if you see the words "build finished" and no errors appear in the Problems window, as shown in Figure 31, the build process has completed successfully.


Figure 31: Successful build console output

If the compilation step ends in an error, please check your project configuration and then ask for help.

## Debugging a project

Once a project has been built, you can load it onto the microcontroller as follows:

1. Connect the development board to the computer. If this is the first time you have connected the board, Windows will install the driver for the debug interface—wait for it to finish before proceeding.
2. Press the Debug ( ) button in the toolbar at the top of the screen. This should load the most recent build of the project onto the microcontroller. If the source code has changed since the last build, it will recompile the project.

If the debug procedure was successful, CCS should enter the "Debug perspective," as shown in Figure 32.

If you see an error reading "Frequency out of range," CCS cannot see your microcontroller. Make sure it is connected and that Windows has finished installing the driver.

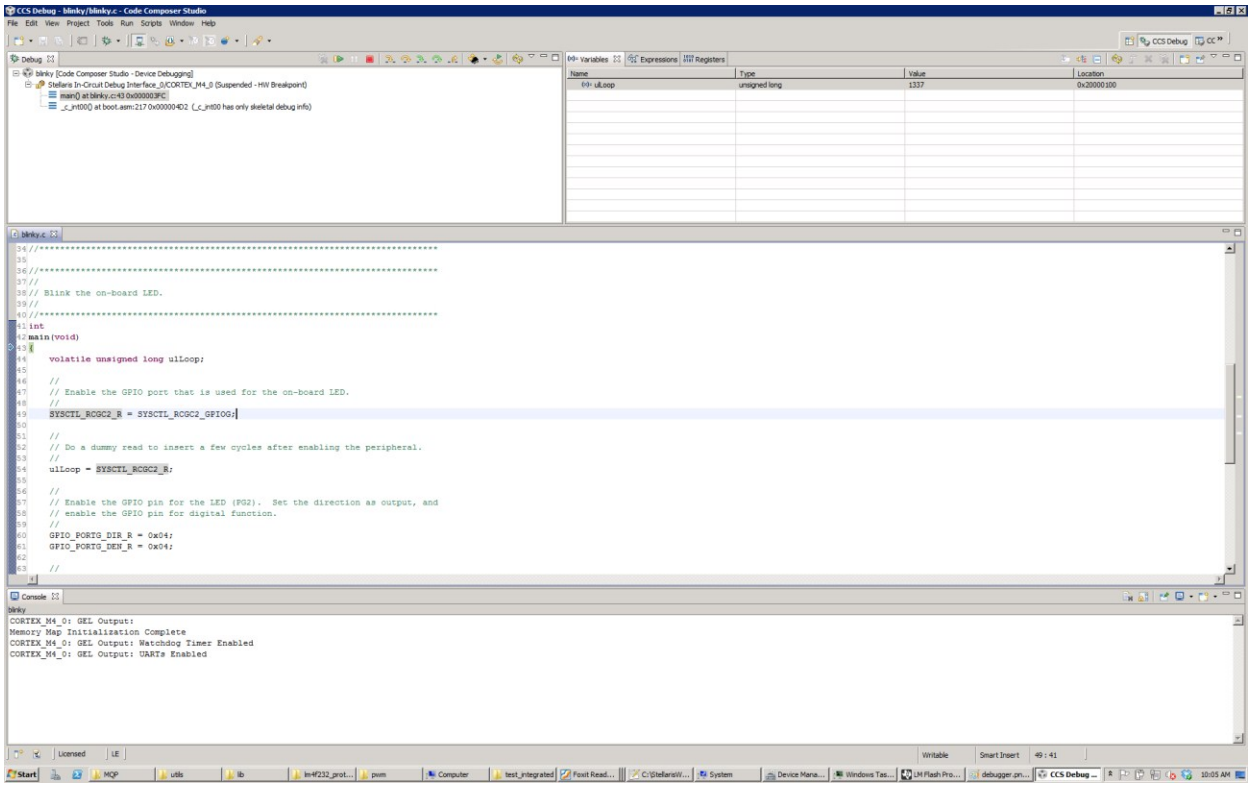



**Figure 32: CCS Debug Perspective**

Like CCS's "Edit" perspective, the Debug perspective displays information in a number of panes, shown in Figure 32, which are described below:

- **Debug Pane**: This pane shows the current device under test as well as the stack trace, which shows a list of the function calls made in the code. Since the code in this example has not yet started running, only main() is in the list.
- **Editing Window:** Just like the "Edit" perspective, the debug perspective shows open source files. Note the line highlighted in green at the start of main: this shows the current position of the program counter in the code, which indicates the next line to execute.
- **Disassembly View:** This shows the assembly instructions for the currently executing code. Note that this pane has a line highlighted in green, just like the editing pane—this shows the currently executing assembly instruction. It is unlikely that you will need to view the assembly instructions to debug your labs, so it is safe to close this pane. You can open it again if you need it.
- **Variables Pane:** This pane shows the value of any variables in the code that are *currently in scope,* which means any variables local to the current function. In this case, the local variable ulLoop can be seen at the start of main() and in this window.

## Running code in the debugger

Once CCS has loaded the code to your microcontroller, it will pause before executing any instructions. You should see an arrow pointing to the first line of code in the project, which is at the start of main(), as shown in Figure 33.

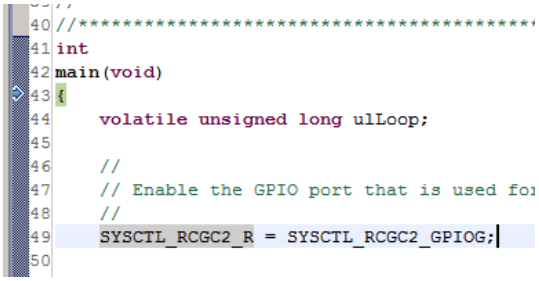



Figure 33: Paused Execution State

This means that the processor is paused and waiting for your input for continuing. By default, it does this when initializing the microcontroller or when it reaches a **breakpoint**, or a defined position in the code for execution to stop. You can control how the microcontroller executes code by using the toolbar in the Debug pane, shown Figure 34.

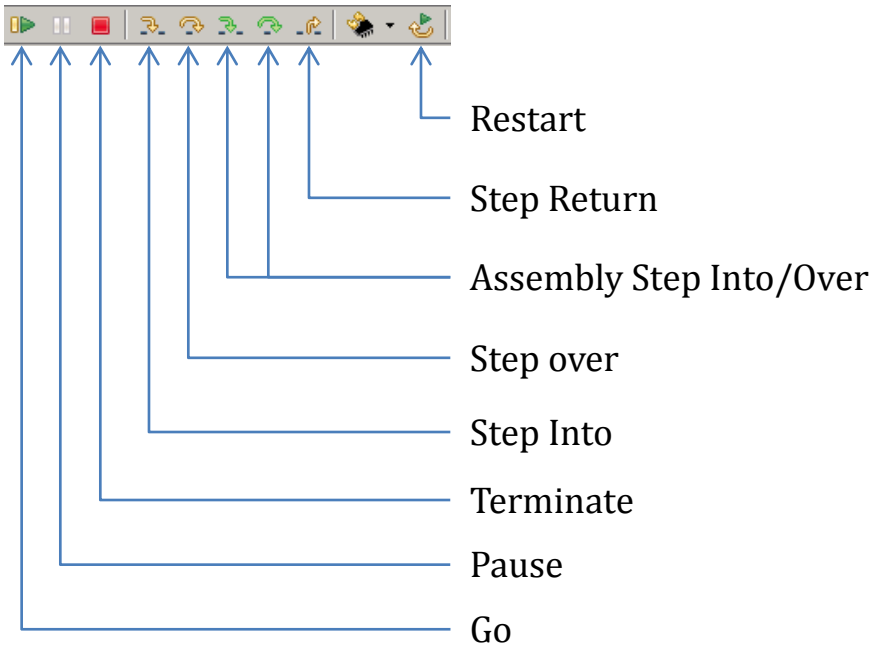



Figure 34: Debug Toolbar Buttons

The debug toolbar can control how the microcontroller behaves in debug mode; the functions available to you are described as follows:

- **Go:** Runs the code until the microcontroller reaches a breakpoint. If no breakpoints are set, this runs your code normally.
- **Pause:** Stops the microcontroller at the current line it is executing. Once paused, you can set breakpoints or examine the current execution state.
- **Terminate:** This halts the microcontroller and executes the debugger. Once the debug operation has terminated, your development board will continue to run without the debugger. This button will return you to CCS's "Edit" perspective.
- **Step Into:** Executes the next line of code denoted by the program counter. If the next line is a function call, the debugger will skip to the function and stop execution at its first line.
- **Step Over:** Executes the next line of code denoted by the program counter. Unlike Step Into, however, the debugger will run any functions called on that line without stopping. This is very useful for debugging code that calls library functions.
- **Assembly Step Into/Over:** These buttons function exactly like Step into and Step over, but they step to the next *assembly* instruction, rather than the next line of C code. It is unlikely that you will need these for your labs.
- **Step Return:** Runs the code until the microprocessor returns from the current function call.
- **Restart:** This restarts the microcontroller's execution state at the first line of main(), as if you stopped the debugger and started it again.

You should test these functions as you debug your labs. You can try a simple use case for setting break points and using a few of these functions as follows:

1. Set a breakpoint on the first statement of code in the editing window by double-clicking on the margin next to the line numbers. You should see a blue circle appear next to the line, as shown in Figure 35.

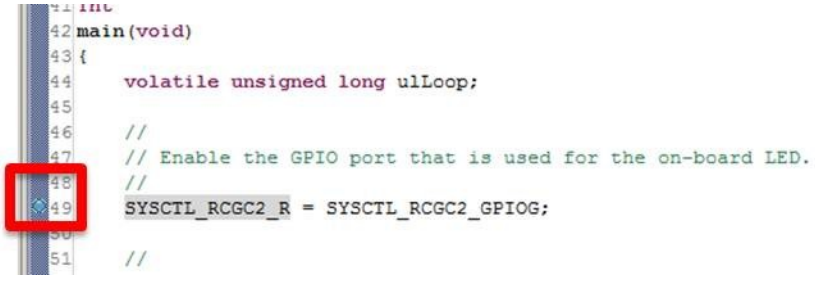



Figure 35: Example breakpoint

**Note:** If the breakpoint circle is grey and not blue, this means that the microcontroller could not set the breakpoint. **The microcontroller can only set breakpoints while the execution state is paused;** pause the debugger before continuing.

2. Press the **Go** button. You should see the program counter jump to the line with the breakpoint and stop.
3. Press the **Step Over** button. You should see the program counter skip to the next line of code.
4. Press **Go** again to allow the microcontroller to run until the next breakpoint. Since the code does not contain any other breakpoints, it will run normally. If you ran the "blinky" project, you should see the LED on your development board flashing on and off.
5. Press the **Pause** button. You should see the editing window shift to another portion of the code—the program counter will point to the line of code where the microcontroller stopped. In the "blinky" project, this line should be in the program's infinite loop inside main().
6. Press the **Terminate** button to exit the debugger. You should see CCS return to the "Edit" perspective and the development board continue to run normally.

This describes only one use case for using the debugging tools. You will become more familiar with the debugger as you complete your labs.

## Viewing variables

Another useful feature of the debugger is its ability to view variables during the code's execution. This is especially important for debugging embedded systems since printf() is not often available (and if it were, it would be so slow that it might introduce bugs in your code!). You can view variables using the **Variables** and **Expressions** panes of the Debug perspective.

## Viewing local variables in the Variables Pane

The Variables pane shows local variables for the current function. This does not include global variables or variables in other functions. When the debugger is paused, it automatically lists these variables in this pane. Figure 36 shows the variables for the "blinky" project at startup (as shown in Figure 35). Since the project only contains one local variable, ulLoop, it is the only variable in the list.

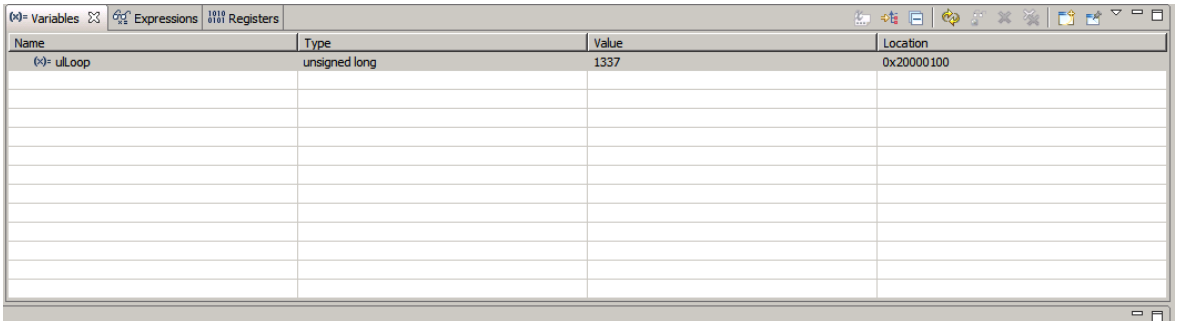



Figure 36: Example variables pane

Note how the window shows the name of each local variable, its type, location in memory, and its value.

## Viewing global and local variables in the Expressions Pane

CCS can also display local and global variables in a single pane based on your preferences. When debugging a project, you can switch to the **Expressions** pane by clicking its tab next to the Variables pane.

**Note**: If you do not see the tab for the expressions pane, you can show it by selecting **Show View > Expressions** from the **Window** menu in CCS.

To demonstrate a use of the Expressions pane, this tutorial will use the example CCS project "interrupts" as an example of a project with multiple local and global variables. The same process applies to any project.

When you have opened the Expressions pane after starting the debugger, it should looks like Figure 37:

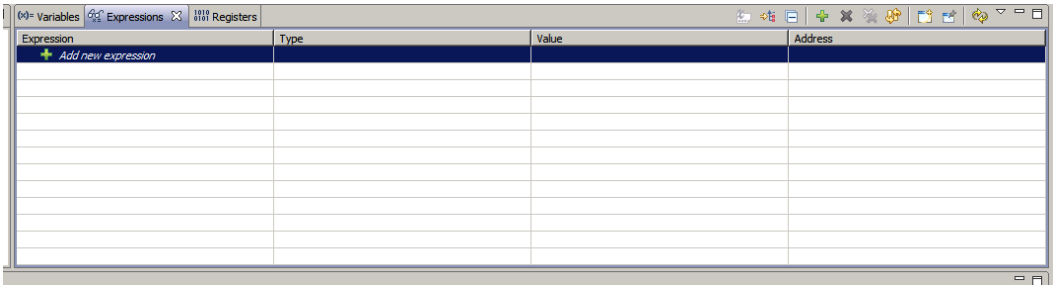



Figure 37: Example expressions pane

By default, the expressions pane does not show any variables. To add them:

1. Click **Add new expression** in the Expressions Pane.
2. Type the name of the variable you want to view.
3. Hit Enter.

In the "interrupts" project, we can try the above for the local variable ulError and the global variable g_ulIndex. After doing so, your expressions pane should look like Figure 38:

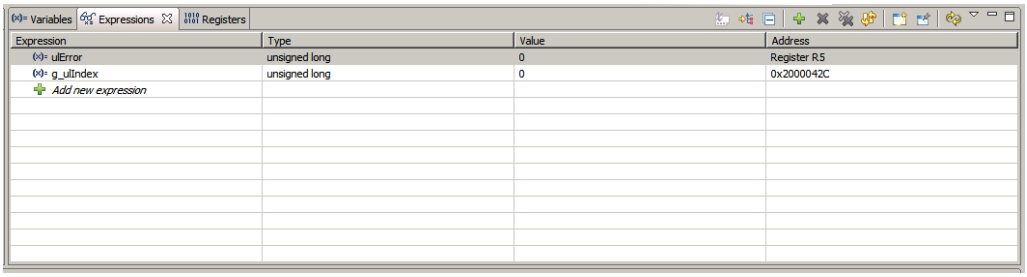



**Figure 38: Expressions pane with variables**

In this case, both ulError and g_ulIndex have a value of 0.

Note that the Expressions pane can only show variables that are *in scope*. This means that it can only view local variables for the function currently executing. Since local variables are allocated on the stack, the debugger has no way to find local variables in other functions (if they are still available in memory).

To test this, you can examine the value of local variable ulError when the debugger enters another function. For the "interrupts" project, you can test this as follows:

1. Set a breakpoint in the function IntGPIOc, which is an interrupt handler.
2. Press **Go** in the debugging toolbar. The microcontroller should run and halt execution in the InitGPIOc function.
3. Examine the expressions window. It should look like Figure 39:

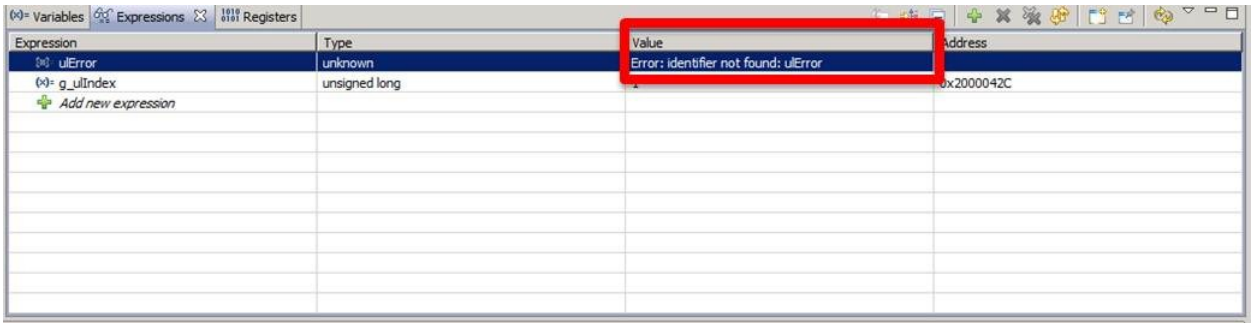



Figure 39: Expressions pane with variable out of scope

Note that the Expressions pane cannot show a value for ulError, which is a local variable inside main(). This is because the debugger is no longer running inside main(), so the name ulError no longer has meaning.

## Changing the display format of variables

You can control the format in which CCS displays variables. To do this:

1. Right click on a variable displayed in the Variables or Expressions panes
2. Select Number Format.
3. Select the format of the variable you want to display, as shown in Figure 40.

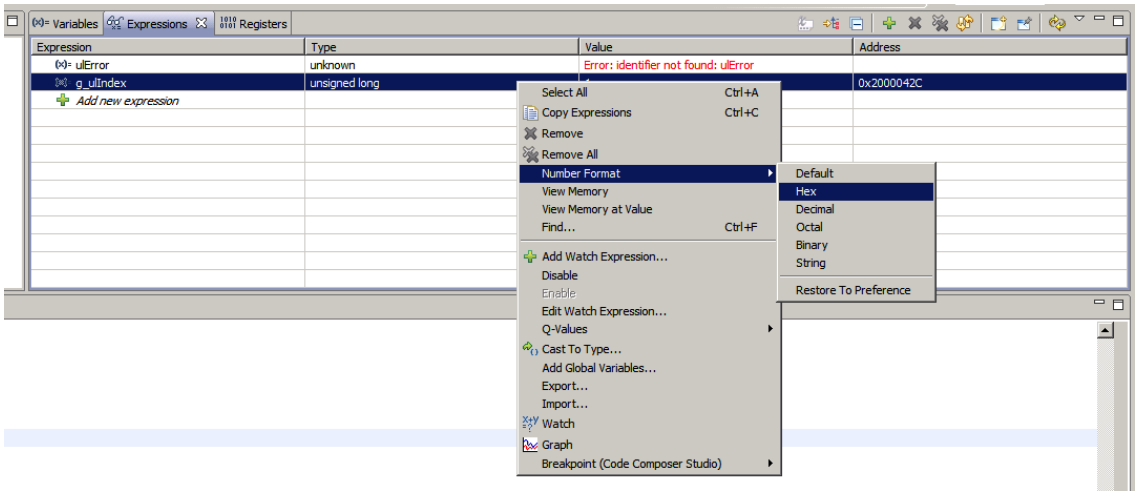



Figure 40: Number format context menu

This allows you to view variables in decimal, hex, or other formats, as may be required by your labs.

## Creating your own projects

Developing your own embedded software for ECE3849 requires creation of your own CCS projects that will build on the microcontroller. This section describes how to create and configure CCS projects to use our microcontroller in lab and include all of the software libraries we will need.

First, create a blank CCS project, as follows:

1. From the menu bar, select **File > New… > CCS Project**
2. Enter a name in the project name field
3. In the Device section, set the Family, Variant, and Connection parameters as follows:
   Family: ARM
   Variant: Stellaris Cortex LM4F23x, Stellaris LM4F232H5QD
   Connection: Stellaris In-Circuit Debug Interface
4. In the Project Templates section, select **Empty project (with main.c)**

Your configuration should look like Figure 41:

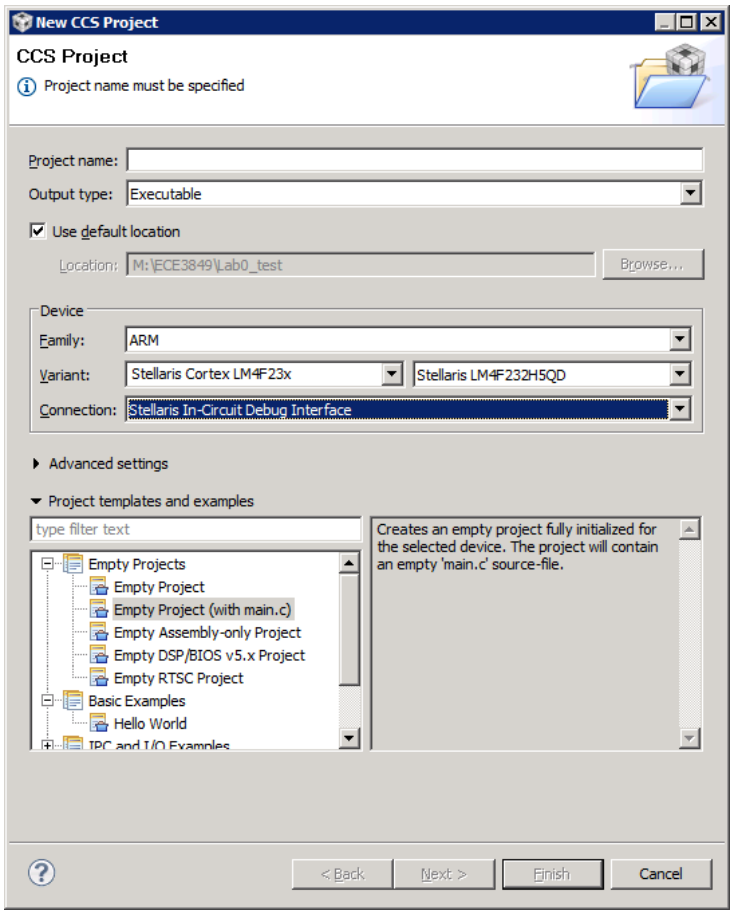



**Figure 41: New Project Window**

5. Click **Finish**.

This creates a skeleton project with a default configuration for the microcontroller we use in lab. In addition, you need to configure the project to use the software libraries we will need for this course. Our labs will utilize the following software libraries:

1. The microcontroller-specific libraries, which contain definitions for addressing the peripherals on the particular chip we use in lab.
2. The StellarisWare libraries (often called driverlib), which provides high-level functions for interacting with the peripherals.
3. The evaluation board-specific libraries, which contain code for the peripherals on our evaluation board.

To add these libraries:

1. Right-click on the new Eclipse project.
2. Select **Properties**.

3. In the pane on the left, select **Build** and click on the tab labeled **Variables**. It may take a moment to load.
4. Create a new variable by clicking **New**. Fill in the fields to match those in Figure 42.

   Use the **Browse** button fill in the path to C:\StellarisWare, rather than typing it, in case your system's configuration is different than the one described here.

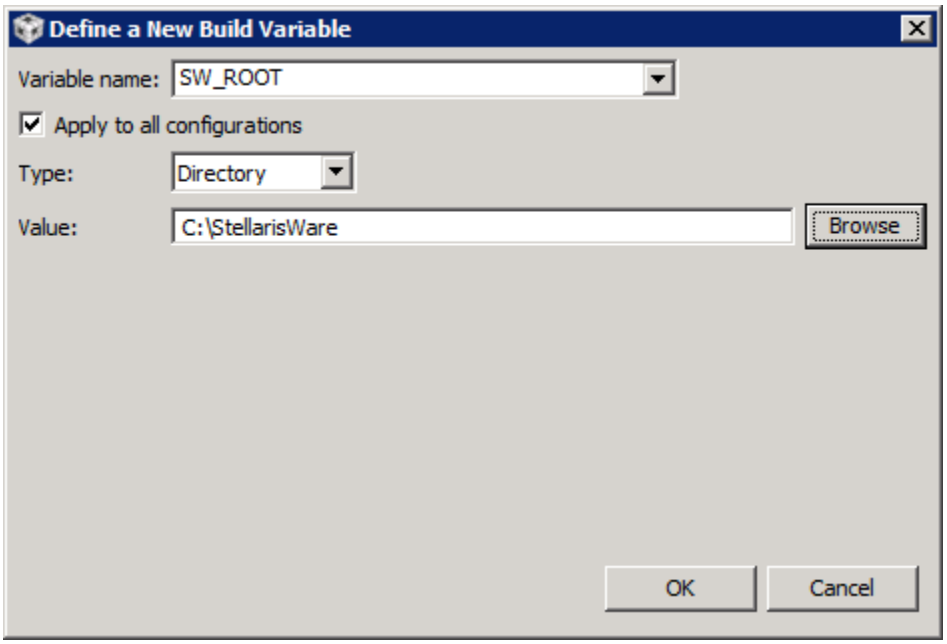



Figure 42: Creating the build variable

5. Name: SW_ROOT
   Location: Click **Browse...** and navigate to **C:\StellarisWare**
6. Click **OK**.

At this stage, your build variables window should look like Figure 43:

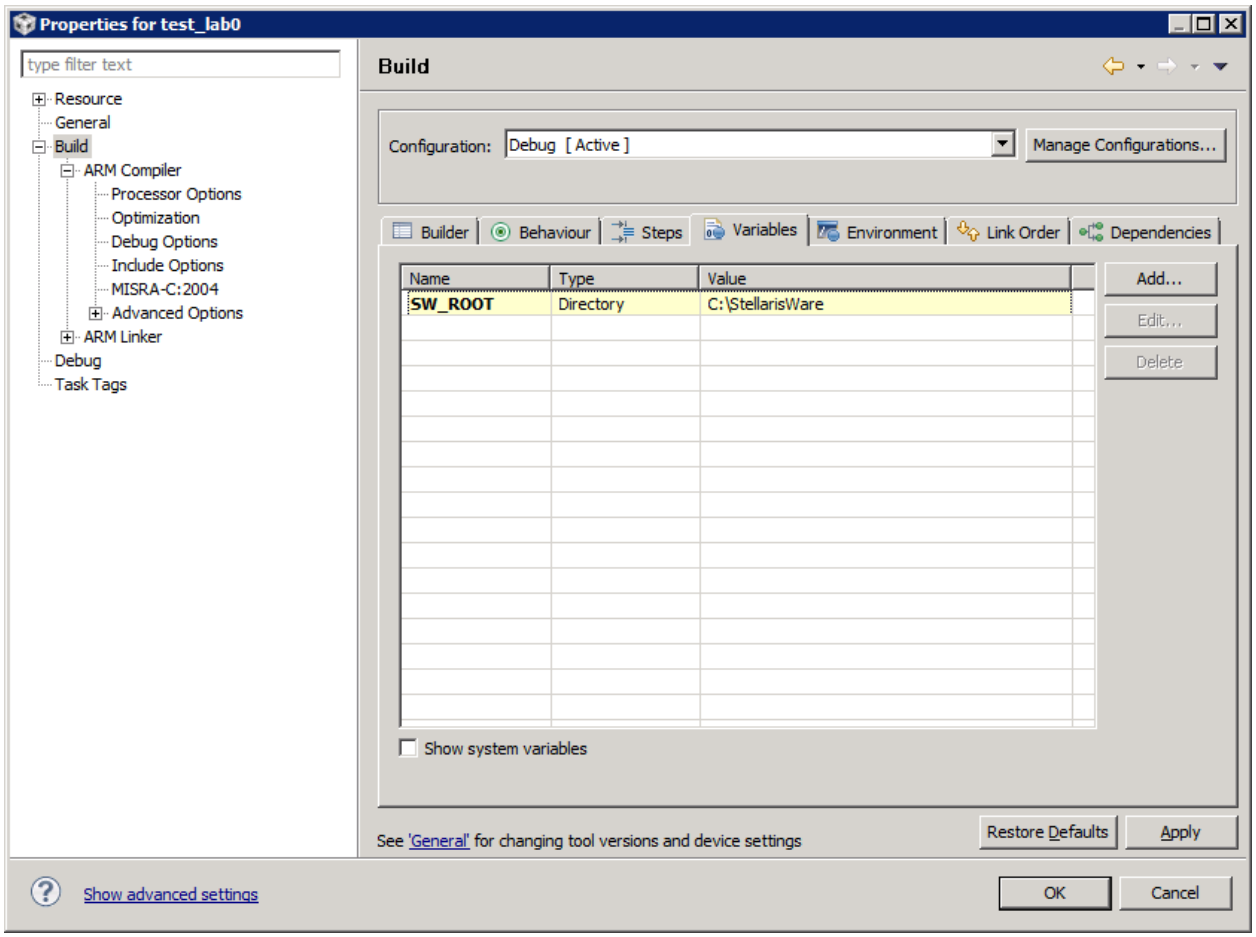



Figure 43: Completed build variables configuration

This variable provides a common way to refer to the directory containing the libraries—this is important because the location of the libraries may change if you use a different machine configuration.

Once the variable has been created, add references to the software libraries. This involves configuring the **Include Path**, or the path the compiler uses to find .h files, and the linker options to include the software library files.

1. From the properties window, expand the tree to find **Build > ARM Compiler > Include Options**
2. Under the pane labeled **-I include path**, click the New button and enter each of the following paths (which are based on the variable created in the previous step), as shown in Figure 44.

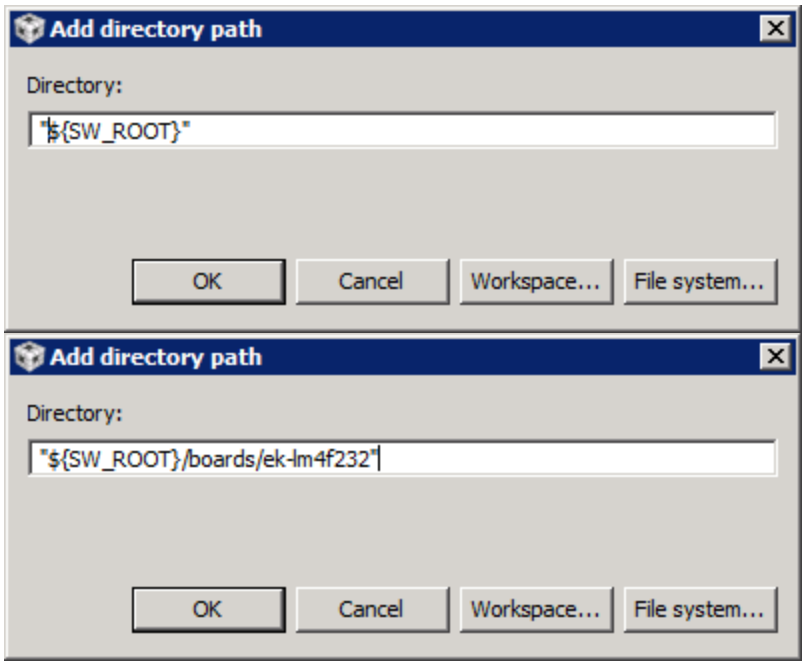



Figure 44: Include paths for StellarisWare libraries

3. When complete, your Include Options window should look like Figure 45.

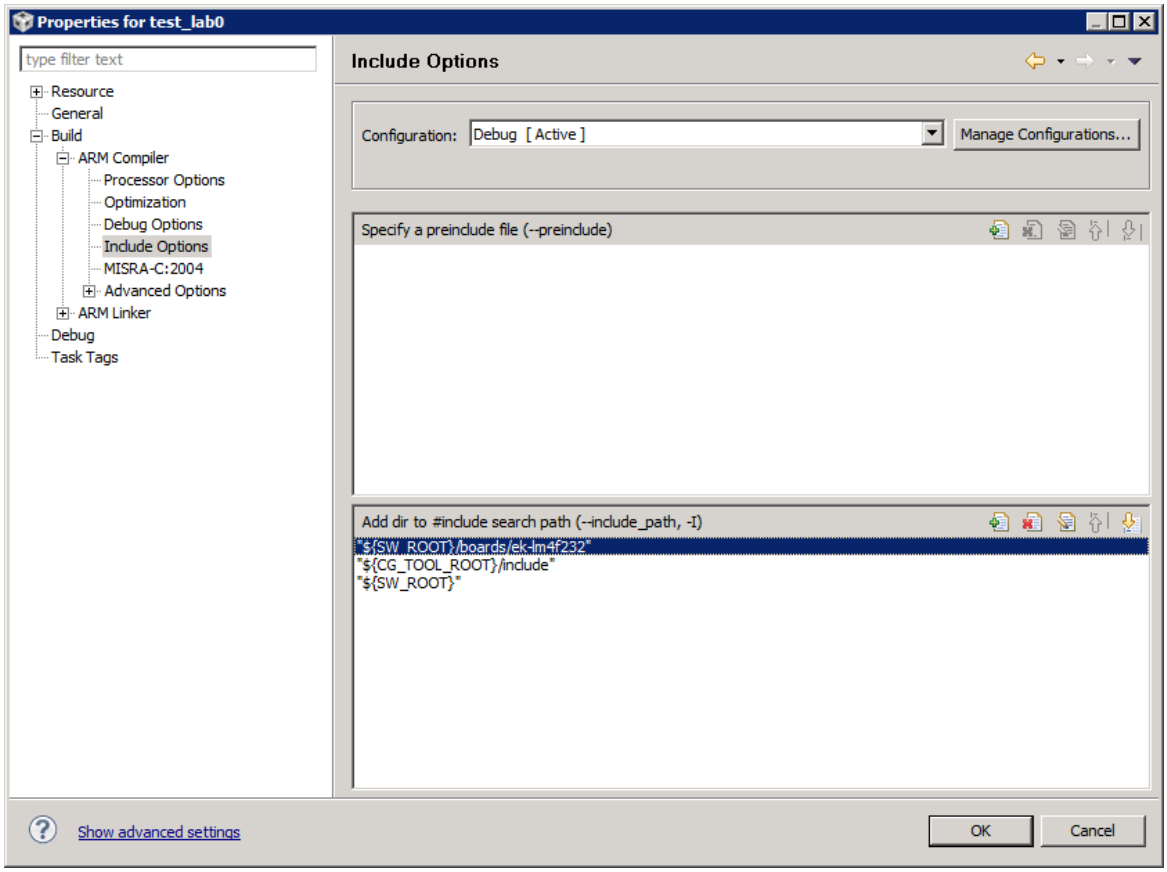



Figure 45: Completed Include Options configuration

4. Click **OK** and reopen the properties window.
5. Expand the tree to find Build > ARM Linker > File Search Path
6. Under the pane labeled **Include library or command file as input –l**, click the New button and enter the path shown in Figure 46 below.

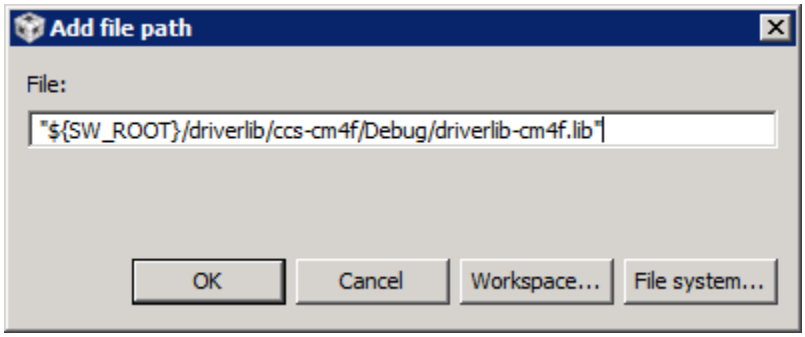



Figure 46: Library path for StellarisWare library

7. When complete, your File Search Path window should look like the example in Figure 47.

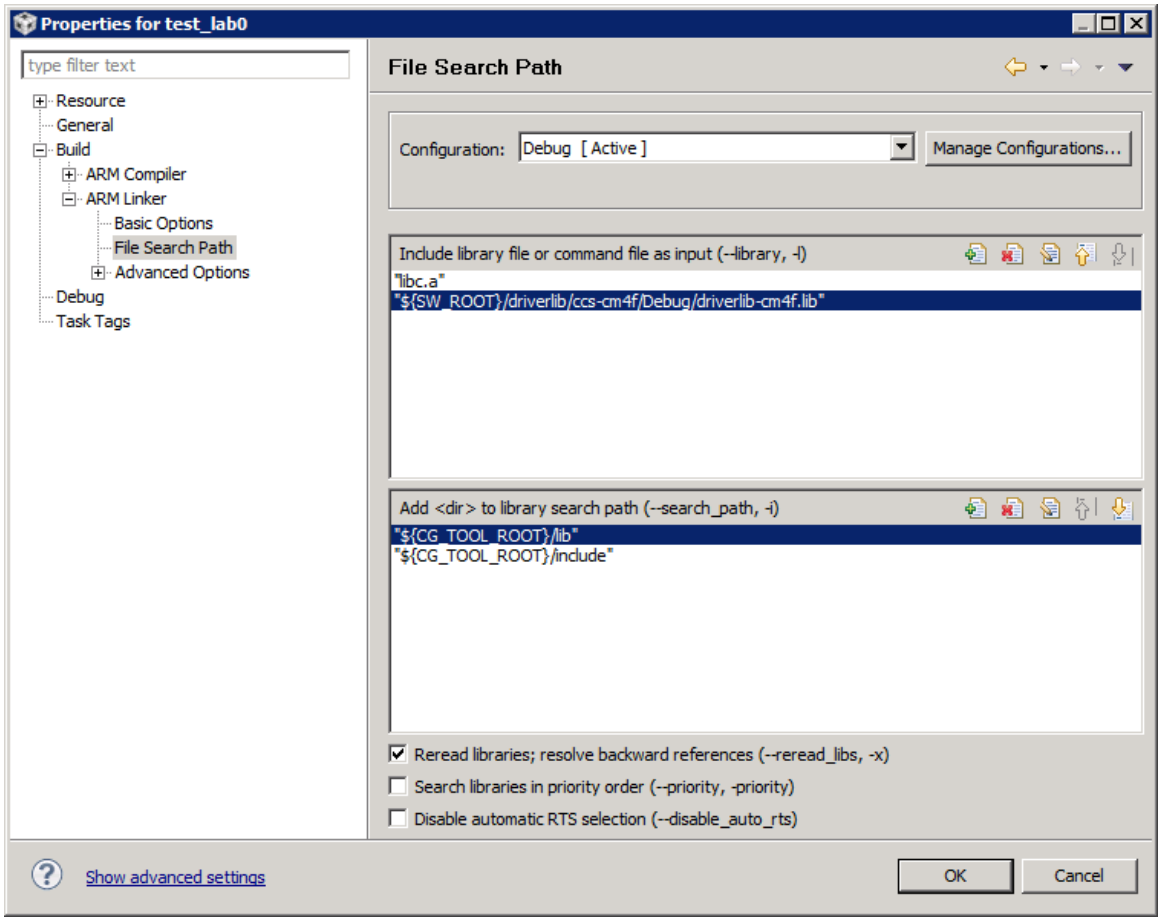



Figure 47: Completed Linker Search Path configuration

8. Click **OK** to close the properties window.

Your project is now configured for using all three software libraries. To verify this, expand the **Includes** line in your project's entry in the Project Explorer pane. You should see three entries, as shown in Figure 48.

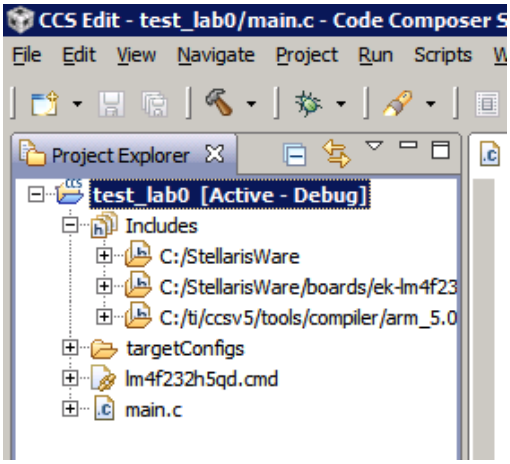



Figure 48: Completed Includes Configuration

You can now build a CCS project using the skeleton code you have been provided in your lab assignment.

## Using the Hardware and Software Documentation

The documentation provided by TI, including the microcontroller datasheet and the StellarisWare driver library, can be very verbose and confusing. Therefore, it can be difficult for a beginning developer to use. However, it is important to learn how to use these documents, as they describe how you can use the microcontroller and software library in your labs. As a professional developer, you will need to understand this kind of documentation.

This section highlights the important points of these documents to use as a starting point for your labs.

### The Microcontroller Datasheet

The datasheet for the microcontroller we use in lab—the Stellaris LM4F232H5QD—is the *de facto* reference for how the microcontroller functions. It describes every function on the device, including every peripheral and register. You can find this extensive document on the course website or on TI's site for the microcontroller.

The document is divided into sections based on the peripherals available on the microcontroller. **Note that the PDF version of the datasheet includes a table of contents to help you find specific sections—your PDF viewer should allow you to use it to jump between the sections.**

Each section contains a lot of information about the available peripherals. When implementing your labs, consider reading the following sections to find the information you need.

### Section Overviews & Functional Descriptions

The beginning of each section has a high-level description of the peripheral device it describes. While this may not seem like important information, it provides a good outline of the functions available in each peripheral and, most importantly, *how the datasheet refers to them in the rest of the document*. This is important for identifying what you need to search for in the rest of the datasheet.

Each section also contains a subsection labeled **Functional Description**. As its name suggests, this section describes how the hardware on the chip functions when provided certain inputs. This can be useful for telling how the registers (and library functions that use them) control the hardware.

### Signal Descriptions

One of the first steps in setting up a peripheral device is configuring the pins it uses on the microcontroller. Every applicable section contains a subsection called **Signal Description**

that explains what pins on the microcontroller connect to the peripheral. You can use this information for performing pin configurations in your labs.

An example signal description for the ADC is shown in Figure 49.

**Table 13-1. ADC Signals (144LQFP)**

| Pin Name | Pin Number | Pin Mux / Pin Assignment | Pin Type | Buffer Type[a] | Description |
|---|---|---|---|---|---|
| AIN0 | 12 | PE3 | I | Analog | Analog-to-digital converter input 0. |
| AIN1 | 13 | PE2 | I | Analog | Analog-to-digital converter input 1. |
| AIN2 | 14 | PE1 | I | Analog | Analog-to-digital converter input 2. |
| AIN3 | 15 | PE0 | I | Analog | Analog-to-digital converter input 3. |
| AIN4 | 144 | PD7 | I | Analog | Analog-to-digital converter input 4. |
| AIN5 | 143 | PD6 | I | Analog | Analog-to-digital converter input 5. |
| AIN6 | 142 | PD5 | I | Analog | Analog-to-digital converter input 6. |
| AIN7 | 141 | PD4 | I | Analog | Analog-to-digital converter input 7. |

Figure 49: Excerpt from ADC Signal Description

Notice how the table shows how each analog channel that connects to the ADC is connected to pins, which are designated by their GPIO reference. (For more information on GPIO references, see Configuring GPIO Pins). For example, analog channel 2 (designated as AIN2) is connected on pin PE1.

This particular table lists the relevant signals for the ADC. If you want to see a list of all of the signals on the microcontroller, see the section at the end of the datasheet labeled **Signal Tables**.

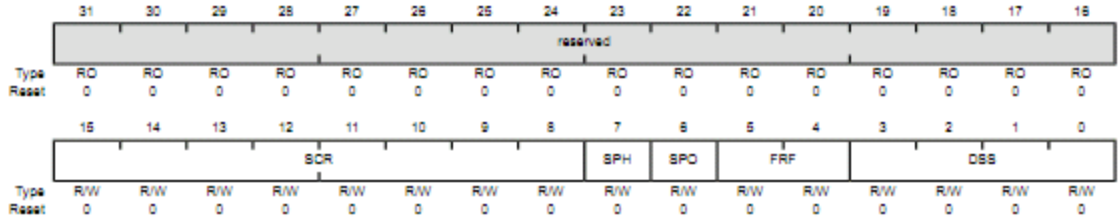
**Note:** Some datasheets contain signal tables for the same microcontroller in different physical packages (for example, a manufacturer might produce a chip in a quad-flag pack (QFP) and a ball-grid array package). Make sure that you read the signal table that corresponds to the chip you are using. **The microcontroller we use in labs is in a 144LQFP** (144-pin Low-profile Quad Flat Pack) package.

## Register Diagrams

All interactions between the hardware and software are controlled by **registers,** or locations in the chip's hardware that can be read or written like a memory device. The microcontroller provides a number of registers for interacting with the on-chip peripherals.

For most of your labs, you will only interact with these using the StellarisWare driver library functions. However, some functions are not provided by the library and must be accessed directly. In addition, you can view the contents of the registers using the **Registers View** in Eclipse when debugging, which can help you figure out how the microcontroller is behaving.

A typical register diagram looks like Figure 50.




| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 31:16 | reserved | RO | 0x0000 | Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation. |
| 15:8 | SCR | R/W | 0x00 | SSI Serial Clock Rate<br>This bit field is used to generate the transmit and receive bit rate of the SSI. The bit rate is:<br>`BR=SysClk/(CPSDVSR * (1 + SCR))`<br>where `CPSDVSR` is an even value from 2-254 programmed in the **SSICPSR** register, and `SCR` is a value from 0-255. |
| 7 | SPH | R/W | 0 | SSI Serial Clock Phase<br>This bit is only applicable to the Freescale SPI Format.<br>The `SPH` control bit selects the clock edge that captures data and allows it to change state. This bit has the most impact on the first bit transmitted by either allowing or not allowing a clock transition before the first data capture edge. |

| Value | Description |
|---|---|
| 0 | Data is captured on the first clock edge transition. |
| 1 | Data is captured on the second clock edge transition. |

**Figure 50: SSI Register Diagram**

This particular register is part of the SSI peripheral, which support SPI transmissions. At the top, it shows how the register is divided into bit fields—each bit field has significance to the peripheral. The bit fields are labeled and described under the diagram.

For example, the field labeled SPH (corresponding to bit 7) indicates how the SSI peripheral will read SPI data. When the value of the bit is a 0, it captures on the first clock edge; when the value is a 0, it captures on the second edge.

The table describing the function of each bit field also lists the following information:

- **Reset Value**: When the peripheral powers up, the bit field will have the value listed here. This indicates the default configuration.
- **Read/Write Mode**: Some fields can only be read (like those indicating chip status), some can only be written (like those that initiate functions); some are read/write (like those that configure peripherals). This field indicates how you can access the given bit field.

Not all bit fields in a register will be useful to your labs. It is important to understand (based on the functional descriptions of the peripherals), which fields matter to your assignment.

## StellarisWare Driver Library Documentation

Most of your labs will interact with the microcontroller using the driver library provided by TI, which is called StellarisWare. This library provides a set of functions that can be used to interface with the microcontroller without accessing the registers directly. This can make your code much more readable and easier to develop.

In addition, the driver library also explains a general procedure for using the functions to configure each peripheral for use. This can be very useful for implementing your labs.

The document is divided into sections based on the peripherals available on the microcontroller. **Note that the PDF version of the datasheet includes a table of contents to help you find specific sections—your PDF viewer should allow you to use it to jump between the sections.**

Consider reading the following sections of the driver library documentation for the peripherals you will use in lab to help find the information you need:

### Introduction

While it may be easy to skip over this section, the introduction provided for each peripheral provides a high-level overview of the functions available to the peripheral. This is a good starting point for figuring out what you need to configure.

### Detailed Description

The name for this section is somewhat misleading, as it is often shorter than the introduction. However, this section provides an overview of which library functions you need to configure a given peripheral. This can help you identify which of the many library functions you need to use for your lab.

### Using the Function Documentation

The remainder of the driver library documentation describes how each library function can be used in your code. An example function description is shown in Figure 51.

#### 4.2.2.23 ADCSequenceDataGet

Gets the captured data for a sample sequence.

**Prototype:**

```
long
ADCSequenceDataGet(unsigned long ulBase,
                   unsigned long ulSequenceNum,
                   unsigned long *pulBuffer)
```

**Parameters:**

***ulBase*** is the base address of the ADC module.

***ulSequenceNum*** is the sample sequence number.

***pulBuffer*** is the address where the data is stored.

**Description:**

This function copies data from the specified sample sequencer output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples. This function only returns the samples that are presently available, which may not be the entire sample sequence if it is in the process of being executed.

**Returns:**

Returns the number of samples copied to the buffer.

Figure 51: Example Function Description

## Understanding Function Prototypes

The function description provides the function's **prototype** and a description of the arguments that you need to pass to the function to use it. The prototype shows the return type (type long), the name of the function, and the function's arguments and their types.

## Function Parameters

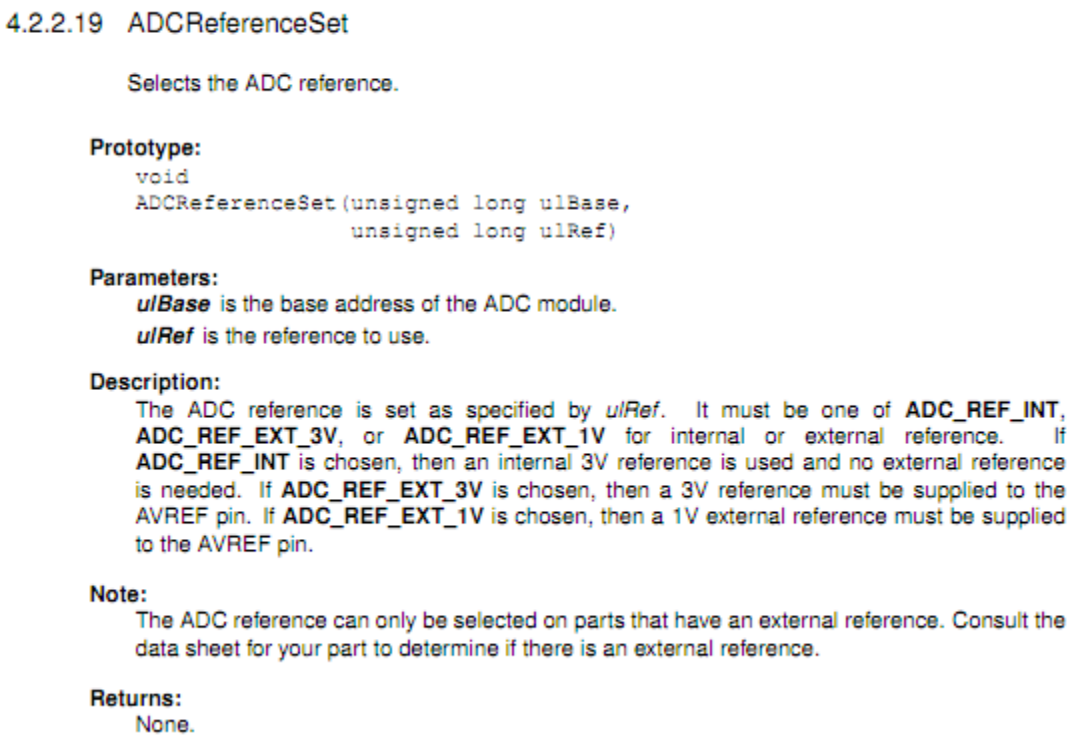
The first parameter to the function is the **base address**. This is common for most of the library functions. Since most microcontrollers have multiple peripherals of the same type, the base address specifies which peripheral to configure.

The driver library contains constants in the code for each base address. These have the format NAMEx_BASE. Where NAME is the name of the peripheral, and x is its number on the chip. For the first ADC peripheral, ADC0, this constant will be called **ADC0_BASE**. For ADC1, it would be **ADC1_BASE**, and so on.

The third argument to the function asks for a pointer to a **buffer**, specified in the document as "the address where the data is stored." In this case, the function will read data from the ADC and *copy it into the buffer provided by the argument*. To explain this differently, the function will "return" the data it is designed to get from the ADC to the given address. This is very common for functions that return large quantities of data.

Finally, the function description also explains what the function returns and provides a bit of information on how the function uses the peripheral.

Some library functions also refer to constants defined by the library, as shown in Figure 52.



4.2.2.19 ADCReferenceSet

Selects the ADC reference.

**Prototype:**

```
void
ADCReferenceSet(unsigned long ulBase,
                unsigned long ulRef)
```

**Parameters:**

***ulBase*** is the base address of the ADC module.

***ulRef*** is the reference to use.

**Description:**

The ADC reference is set as specified by *ulRef*. It must be one of **ADC_REF_INT**, **ADC_REF_EXT_3V**, or **ADC_REF_EXT_1V** for internal or external reference. If **ADC_REF_INT** is chosen, then an internal 3V reference is used and no external reference is needed. If **ADC_REF_EXT_3V** is chosen, then a 3V reference must be supplied to the AVREF pin. If **ADC_REF_EXT_1V** is chosen, then a 1V external reference must be supplied to the AVREF pin.

**Note:**

The ADC reference can only be selected on parts that have an external reference. Consult the data sheet for your part to determine if there is an external reference.

**Returns:**

None.

**Figure 52: Example function description using constants**

The description indicates that the parameter ulRef must be one of **ADC_REF_INT**, **ADC_REF_EXT_3F**, or **ADC_REF_EXT_1V**. The names listed in bold refer to **constants** defined in the library. The actual value of the constants does not matter to you—it's merely a reference to select a particular configuration in the library function. (The values actually correspond to the bit configurations to the chip registers, but this does not matter when calling the function.)

To call a function using one of these constants, simply type the name of the constant as an argument. Some functions will accept more than one of these constants concatenated with a bitwise OR—the description will indicate when this is the case.

For example, you could call this function as follows:

```
ADCReferenceSet(ADC0_BASE, ADC_REV_EXT_3V);
```

# Using GPIO Peripherals

This section describes the basic functionality of General Purpose Input/Output (GPIO) peripherals on the Stellaris microcontroller. It is intended to provide a starting point for labs you may encounter that require devices that use GPIO pins. This section will discuss the basic functionality of GPIO pins, how to configure them on the microcontroller, and how to access additional information that may prove useful to your lab implementation.

## Basic functionality

A full description of how GPIO pins function is beyond the scope of this document. However, on its most basic level, GPIO (also called Digital I/O) is a way for a microcontroller to read a logical 1 or 0 on an input pin, or output a logical 1 or 0 to an output pin. This it makes it possible to read from simple inputs like switches, drive devices like LEDs, or implement communications protocols.

GPIO pins are commonly used to implement simple devices like buttons or switches on the microcontroller. Consider the switches on the development board as shown in Figure 53 (and also found in the LM4F232 development board's user guide).

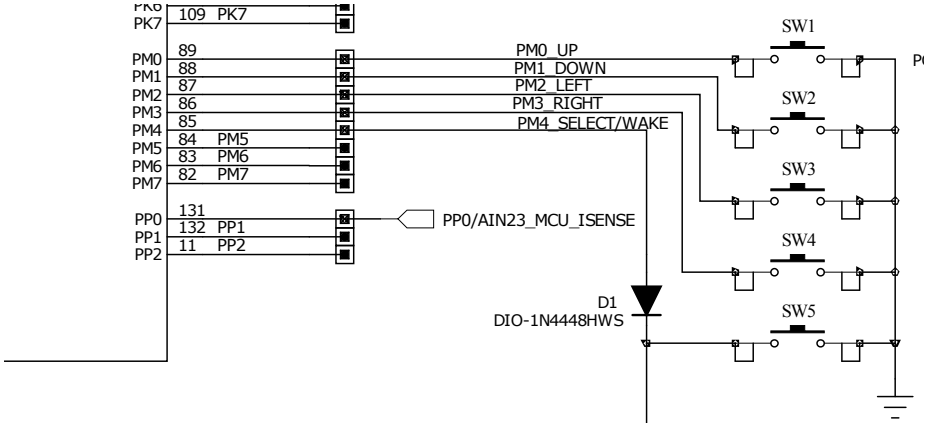



**Figure 53: Directional switches on the LM4F232 Development board**

Here, the switches are shown connected to the microcontroller on the left, which provides an internal pull-up resistor (described in detail in the next section). When the switch is not depressed, the GPIO pin connected to it reads a logical high. When the switch is depressed, it completes a circuit from the microcontroller to ground, so the pin will read a logic low.

## GPIO Identifiers

All GPIO pins on Stellaris Microcontrollers (like the ones you use in lab) are given an identifier. The GPIO pins corresponding to the switches shown here have identifiers PM0-PM4. The identifier can be broken down as follows:

- **P**: This stands for "Port," referencing how GPIOs are allocated on the chip
- **M**: This is the GPIO pin's **Port Group** on the microcontroller, denoted by a letter. In this case, the pins belong to port M.
- **0-4**: Each GPIO pin is assigned a number on its designated port group, which indicates how it can be identified in the code. This corresponds to the bit each pin occupies in the GPIO port's data register.

## Accessing GPIO Ports

Each GPIO port is connected to a data register on the microcontroller with the name of **`GPIO_PORTx_DATA_R`**, where x is the letter corresponding to the port group. For port M, the register would be **`GPIO_PORTM_DATA_R`**.

**Note**: You cannot read to or write from GPIO pins without configuring them first. This is described in the next section. This section is important to discuss first because it explains how GPIO pins work.

### Accessing GPIO registers

The data register can be used to read or write data to GPIO pins. You can do this by reading or writing to the register just like another variable in the code.

For example, you could write to the entire GPIO port (controlling all 8 bits), as follows:

```
GPIO_PORTM_DATA_R = 0x00;
```

This would write a '0' to each GPIO pin on Port M, which would drive each pin to 0V.

Or read from it as follows:

```
unsigned long ulOutput = GPIO_PORTM_DATA_R;
```

This would read from each GPIO pin on Port M—the result would be loaded into the variable **`ulOutput`**.

The important point to keep in mind is that each bit in the register (and all other registers for GPIO ports) corresponds to a specific GPIO pin. To understand this, we can examine the microcontroller's datasheet, which shows the data register as in Figure 54 (p. 656 of the datasheet, emphasis added).

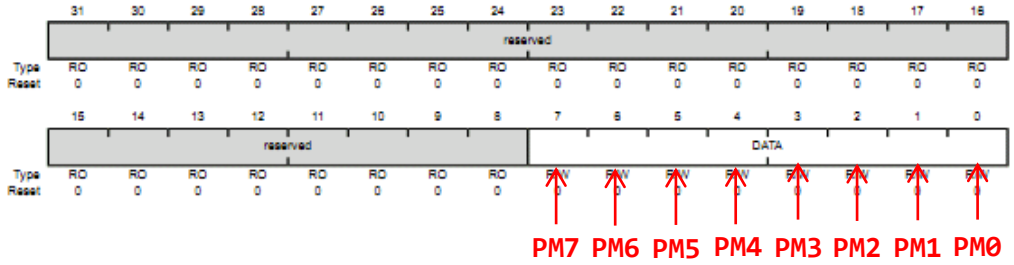



Figure 54: GPIO Data Register

According to the register diagram, only the lower 8 bits of the register are valid. These bits correspond to the 8 pins available on each GPIO port. Each bit corresponds to a specific pin on the GPIO port, starting with the least significant pin, as indicated in the figure.

## Reading and writing to specific GPIO pins with bit masking

There are many ways to use GPIO pins. The canonical way is to read the data register and use **bit masking** to obtain only the bits you need. You can also use the StellarisWare library functions **`GPIOPinRead()`** and **`GPIOPinWrite()`**. These are described in the StellarisWare example code and in the StellarisWare driver library documentation.

The procedure for bit masking is simple: when reading a register, perform a bitwise AND with only the bits you need. For example, if you wanted to read bits 0-4 on port M (corresponding to the switches on the board), you could write:

```
unsigned long ulButtons = GPIO_PORTM_DATA_M & 0x0f;
```

This operation would clear all bits in the register that are not bits 0-4 (try it on paper). Since we only wanted to read the value of bits 0-4, this is what we need.

## Configuring GPIO Pins

GPIO stands for General Purpose Input and Output, meaning that GPIO pins can perform a very large number of functions. Therefore, you need to configure each GPIO pin to suit your needs before you can use it. This document will provide an overview of the common options available and how you can use them.

### GPIO Pin Types

Most pins on the microcontroller (and every one available on the development board) is **multiplexed** with a GPIO pin. This means that each pin can serve a number of functions—one of which is GPIO. The other functions can be for peripherals like the ADC, Timers, or other devices and are generally called **alternate functions**.

When the microcontroller starts up, you will need to configure the pin type. For GPIO pins, there are two possible pin types:

- **GPIO Inputs**: Inputs allow you to read digital values connected to the pins
- **GPIO Outputs**: Outputs allow you to drive the pin to a logical high or a logical low to provide an output for simple devices

You can configure the pin type using the functions **`GPIOPinTypeGPIOOutput()`** or **`GPIOPinTypeGPIOInput()`**, as described in the StellarisWare Libraries.

To figure out how to use one of these functions, we can look at its signature as shown in the documentation, which looks like:

```
void GPIOPinTypeGPIOInput(unsigned long ulPort, unsigned char ucPins)
```

The first argument to this function is the GPIO port you want to configure. The driver library defines names for each peripheral on the microcontroller (including GPIO ports) so you can refer to them in their library functions. These names refer to the **base address** of the peripheral. For the GPIO ports, they are named **GPIO_PORTx_BASE,** where X corresponds to the GPIO port. If we were using Port M, this would be **GPIO_PORTM_BASE.**

The second argument needs to know which pins on the given port should be configured. According to the documentation, you should provide the "bit-packed representation of the pins," which simply means the bitwise OR of the pins you want to configure. The driver library also provides names for each GPIO pin, so we can use these as well. For example, if you wanted to configure pins PM0-4 as inputs, you could provide the argument

```
GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4
```

Therefore, the entire function call would look like:

```
GPIOPinTypeGPIOInput(GPIO_PORTM_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 |
GPIO_PIN_3 | GPIO_PIN_4);
```

### Pad Configurations

In addition to configuring the pin type, GPIO pins have a few other options to keep in mind. An exhaustive demonstration of how to use these is well beyond the scope of this document, but it is important to know that these configurations exist. For more information, look in the GPIO section of the driver library documentation, or the microcontroller datasheet.

Most of these configurations can be set with the **`GPIOPadConfigSet()`** function in the driver library.

Some of the most useful pad configurations are as follows:

- **Weak pull up/pull down**: Each GPIO pin contains a **tiny** pull-up or pull-down resistor inside the microcontroller, which can be enabled if you use this option. These resistors are very weak, but are useful for implementing devices like switches.
- **Open-drain:** An open-drain pin can be driven to a logical low, or can act like it is disconnected (high impedance)—it cannot drive a logical high. This can be useful in some configurations, which you will encounter in your labs.
- **Current rating:** By default, each GPIO pin can source or sink 2mA of current. You can configure a certain number of GPIO pins to provide up to 8mA with certain settings.

### Further Reading

For more information on how to use GPIO pins, consider reading the following documents

- **StellarisWare Driver Library Guide**: Section 12 describes how to use the library functions in a lot of detail.
- **Microcontroller Datasheet:** Section 10 of this document explains the available GPIO registers and how each GPIO peripheral option works.
- **Peripheral Example code:** The example code does not provide any specific GPIO examples, but most of the projects utilize GPIO pins in some way—any of these can provide good examples.

For more information on how to find these documents, see Useful documentation and examples.

## Useful documentation and examples

As you develop code for your labs and use peripherals on the microcontroller and development board, the following documents may prove useful to your lab implementations:

If you use any of these resources to develop your lab—especially the code examples—be sure to cite this in your code and lab report.

### TI LM4F232 Evaluation Board Manual

Location: Course website

This document describes the capabilities and features provided by the TI evaluation board containing the microcontroller, including the schematics and pin descriptions. This document is useful for determining how peripherals are connected to the evaluation board.

### Stellaris Peripheral Driver Library User's Guide

Location: Course website, C:\StellarisWare\docs\SW-DRL-UG-9453.pdf on lab machines

This manual describes the library functions that TI has provided for interfacing with the microcontroller, including functions for controlling all of the on-chip peripherals like the timers, interrupt controller, and GPIO pins. It provides detailed descriptions of the signatures and valid inputs for each function as well as high level explanations of each peripheral can be configured using the functions. If you are trying to configure a peripheral with the library functions, check here before checking the datasheet, as it usually provides more friendly explanations.

### StellarisWare Example Peripheral Code

Location: C:\StellarisWare\examples\peripherals on lab machines

StellarisWare includes a wealth of example code projects to demonstrate sample applications of the peripherals available on Stellaris microcontrollers. This directory contains simple and well-commented examples that demonstrate *one possible way* to set up peripherals like the ADC, SSI controller, etc. These examples provide a good starting point for learning the **syntax and general procedure** of interfacing with peripherals: it is unlikely that the example code will use the same configuration as your design requires, so **do NOT copy and paste it into your project**.

Instead, read through the code and the comments, referring to the library documentation as necessary, to figure out how the code works.

### StellarisWare Example Projects for LM4F232 Evaluation Board

Location: C:\StellarisWare\examples\boards\ek-lm4f232

This directory contains a set of CCS projects that demonstrate certain peripherals on the TI evaluation board used in lab. Like the example peripheral code, these projects are well-documented and can provide a good reference for using the on-chip peripherals. These projects will work if imported directly into CCS (as described in this guide); however, their implementation is often more complicated than the examples.

### Stellaris LM4F232H5QD Datasheet

Location: Course website

This extensive document provides verbose descriptions of all the features provided by the microcontroller. It also includes descriptions of every register available for access. This document is very verbose, but provides the most complete explanation of how to use the on-chip peripherals.

# Troubleshooting

This section describes some common errors you might encounter when writing software for your labs and potential solutions. It is **NOT** intended to be an extensive troubleshooting guide. For the errors listed, it describes a potential solution or suggestions for investigating the problem further.

## Building Your Code: Cannot Open Source File

### Symptoms

When building your code, you may see an error in the console starting with "**cannot open source file <file.h>**"

### Recommended Solution

This usually indicates a problem with your File Search Path. Check your settings for the project configuration in Starting CCS for the first time.

## Debugging Your Code: Cannot Open Source File

### Symptoms

When debugging your code and pressing the "Suspend" button, the editing window opens a new tab with the following error, as shown in Figure 55.

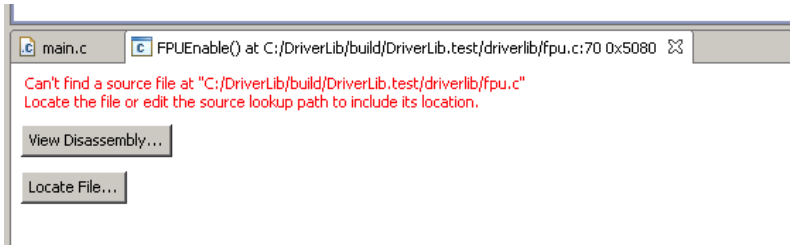



**Figure 55: Debugger Cannot Find Source File**

### Recommended Solution

This is not an error. When the code is suspended while it is running one of the library functions (note that the figure indicates the current function is FPUEnable(), a library function), it will attempt to load the source code for the function to display it in the debugger. However, the library functions are linked into the project as a pre-compiled binary, so the debugger cannot find the source code for it.

To halt the debugger in your code, set a breakpoint at some position in your file, or click **Step Return** until it stops in your code. If the **Step Return** option is not available, click **Assembly Step Next** until it appears.

## Errors Encountered During Linking: Unresolved Symbols Remain

### Symptoms

When building your project, you see an error like the one in Figure 56. The functions listed as undefined are from the StellarisWare library.

```
undefined              first referenced
 symbol                    in file
---------              ----------------
FPUEnable              ./main.obj
GPIOPinConfigure       ./tmp102.obj
GPIOPinTypeI2C         ./tmp102.obj
GPIOPinTypeI2CSCL      ./tmp102.obj
GPIOPinTypeUART        ./main.obj
I2CMasterBusy          ./tmp102.obj
I2CMasterControl       ./tmp102.obj
I2CMasterDataGet       ./tmp102.obj
I2CMasterDataPut       ./tmp102.obj
I2CMasterInitExpClk    ./tmp102.obj
I2CMasterSlaveAddrSet  ./tmp102.obj
I2CSlaveIntClear       ./main.obj
IntMasterEnable        ./main.obj
SysCtlClockGet         ./tmp102.obj
SysCtlClockSet         ./main.obj
SysCtlDelay            ./main.obj
SysCtlPeripheralEnable ./tmp102.obj

error #10234-D: unresolved symbols remain
error #10010: errors encountered during linking; "test_TMP102.out" not built

>> Compilation failure
gmake: *** [test_TMP102.out] Error 1
gmake: Target `all' not remade because of errors.

**** Build Finished ****
```

Figure 56: Unresolved Symbols Remain

### Recommended Solution

This commonly indicates a problem with your Linker Search Path settings. Check your project configuration against the steps described in Starting CCS for the first time.